

初中生C++竞赛宝典

数据结构实战



J组专用辅导书

初中生C++竞赛宝典

数据结构实战



主 编：洪嘉璐
副 主 编：张书颇 陈兴佳

本册主编：洪嘉璐
编写人员(按姓氏笔画排列)：
陈兴佳 李舒红 张书颇 田小琴 廖昌升

信息技术作为数字时代的核心驱动力，正深刻重塑人类社会的生产、生活与思维方式。在全球科技竞争日益激烈的背景下，信息技术不仅是国家创新能力的关键支柱，更是青少年面向未来必备的核心素养。我们编写了《初中生C++竞赛宝典：数据结构实战（J组专用）》，旨在通过系统化的知识体系与实战化的训练模式，助力学生夯实编程基础、发展计算思维、提升算法设计与问题解决能力。

本教材严格遵循中国计算机学会（CCF）CSP-J/S竞赛大纲要求，结合初中生认知特点与竞赛实战需求，聚焦数据结构这一核心模块，构建了从基础理论到高阶应用的完整学习路径。教材内容涵盖线性结构、树形结构、树形拓展及图论基础四大篇章，通过层层递进的知识脉络、经典赛题解析与拓展训练项目，帮助学生实现从“知识理解”到“竞赛实战”的跨越。

教材特色鲜明，体现以下核心理念：

1. 体例创新——以素养为导向，构建分层学习框架

教材采用“概念导引—实例剖析—代码实现—真题演练”的四维结构，每章设置“思维导图”“易错辨析”“知识图谱”等特色栏目，支持学生通过自主探究、合作讨论与项目实践深化理解，为计算思维与创新能力的培养提供结构化支持。

2. 内容精准——聚焦竞赛核心，强化专业性与实用性

精选链表、栈、队列、二叉树、字典树、并查集、图论等CSP-J高频考点，以C++语言为载体，结合STL库实战技巧，剖析数据结构的设计原理与应用场景。所有案例均经过竞赛命题专家审核，确保概念表述严谨、代码规范高效。

3. 能力进阶——设计梯度任务，激活高阶思维

通过“基础巩固”“综合应用”“挑战提升”三级习题系统，引导学生从模仿实现走向创新设计。特别设置专题讨论，培养严谨的工程思维与系统性解决问题的能力。

4. 案例多元——融合学科视野，彰显技术价值

教材案例涵盖社交网络分析、路径规划、字符串处理等真实场景，融入人工智能、物联网等前沿领域背景，帮助学生理解数据结构技术在社会发展、科学中的多维价值，激发学习内驱力。

5. 技术贯通——注重原理迁移，突破工具局限

以C++语法为切入点，深入揭示数据结构与算法的通用设计范式。通过对比不同存储方式（如邻接矩阵与邻接表）的时空效率，引导学生理解技术选型的底层逻辑，培养“透过代码看本质”的抽象思维能力。

本册教材为初中生C++竞赛专项辅导用书《数据结构实战（J组专用）》。编写初衷是助力同学们在信息学竞赛的起点阶段夯实基础，通过系统学习数据结构的核心知识与应用技巧，逐步掌握竞赛解题的“思维武器”，为冲刺更高目标奠定坚实根基。

在信息学竞赛的赛场上，数据结构是破解题目的“利剑”，更是优化算法效率的“密钥”。本书围绕J组（普及组）竞赛要求，精选链表、栈、队列、树、图等基础数据结构，结合递归、遍历、动态规划等经典算法，通过“知识讲解+真题拆解+实战演练”的三维模式，帮助同学们实现从理解概念到独立编程的跨越。期待大家在学习后不仅能清晰描述每种数据结构的特性与操作，更能针对竞赛题目灵活选择工具，编写出高效优雅的代码，在解题过程中初步形成算法设计与优化的意识。

本册教材在编写过程中得到了各方面的大力支持。北京大学计算机系李晓明教授、浙江大学计算机学院卜佳俊教授和翁恺教授、北京航空航天大学欧阳元新副教授在百忙之中对书稿内容进行了审阅。胡金锦、赵军、周海君三位高中教师对书稿内容提出了宝贵的修改意见。

由于水平有限，本书可能还存在不足之处。希望大家在教材使用过程中，能够及时将意见和建议反馈给我们，对此，我们深表谢意。

目 录

MULU

第一章

线性结构：链表、栈与队列

1.1 链表.....	2
1.2 栈.....	13
1.3 队列.....	28

第二章

树：树与二叉树、字典树

2.1 树的基本概念.....	42
2.2 二叉树.....	51
2.3 字典树.....	60

第三章

树的应用：二叉搜索树与并查集

3.1 堆.....	73
3.2 并查集.....	81
3.3 最近公共祖先.....	87

第四章

图论初步：存储与遍历

4.1 图的基本概念.....	98
4.2 图的存储与遍历.....	107
4.3 最短路.....	119

第一章

线性结构：链表、栈与队列

当你走进一间杂乱无间的房间，找一本书可能要翻遍每个角落，这样不仅浪费时间，还损耗精力。但要是合理分类，比如把课本放一起、漫画放一起，找起来就轻松多了。数据结构就像是你整理房间的方法。在编程里，数据结构决定了如何存储、组织和管理数据，让程序能高效地运行，就像有序的房间让生活更便捷。而这一章，便是我们启程的关键站点——数据结构的基础地带，重点聚焦链表、栈和队列。

本章将借助经典的竞赛真题，带领大家深入掌握链表、栈和队列这些数

据结构的基本用法。在解题过程中，你会真切感受到链表如何像灵活的书架布局，动态调整图书存放位置；栈怎样类似整理书籍时，处理复杂逻辑；队列又如何像图书馆借阅排队，有序调度资源。通过对这些真题抽丝剥茧般的分析，详细的代码实现演示，大家不仅能透彻理解各类数据结构在实际问题中的运用，更能举一反三，面对不同竞赛题目时，精准判断并选择最合适的数据结构来存储和处理数据，为在 CSP 竞赛中取得优异成绩筑牢根基。



1.1 链表——更好地插入和删除

本节学习目标

- ◆ 了解链表与数组的区别，能够对比链表与数组在插入、删除操作上的性能差异。
- ◆ 掌握链表的插入、删除和遍历操作，理解链表的基本操作和通用方法。
- ◆ 理解链表操作的时间复杂度，并能够解释其优势。
- ◆ 识别适合使用链表解决的问题场景，独立设计链表解决方案实际编程竞赛中的问题。
- ◆ 能够通过链表的应用，提升逻辑思维和问题解决能力。

1.1.1 问题引入：约瑟夫环

问题描述

n 个人围成一圈，从第一个人开始报数，数到 m 的人出列，再由下一个人重新从1开始报数，数到 m 的人再出圈，依次类推，直到所有的人都出圈，请输出依次出圈人的编号。

具体实例

假设有5个人（ $n=5$ ），每次报到3的人出列（ $m=3$ ），过程如下：

在第一轮中，从1开始报数，数到3时3出列，剩下[1, 2, 4, 5]；第二轮从4开始报数，数到3时1出列，剩下[2, 4, 5]；第三轮从2开始报数，数到3时5出列，剩下[2, 4]；第四轮从2开始报数，数到3时2出列，剩下[4]；最后一轮从4开始报数，数到3时4出列。最终出列顺序为[3, 1, 5, 2, 4]。

1.1.2 知识核心

我们来思考一下用数组解决这个问题的做法。

创建一个长度为 n 的数组，用数组元素来表示每个人，通过标记的方式来表示某人是否已经出圈。在报数过程中，不断遍历数组，当数到 m 时，将对应的数组元素标记为已出圈，然后继续从下一个未出圈的元素开始报数。

虽然数组可以解决这个问题，但它存在一些明显缺陷。在每次有人出圈后，后续的报数需要跳过已经出圈的人，这就需要不断地遍历数组，判断元素是否已经出圈，导致时间复杂度较高。而且，数组的插入和删除操作相对复杂，对于这种动态变化的问题，使用数组会显得不够灵活。这时候我们可以考虑用链表来解决这个问题。

在此，先回顾一下链表的重要知识。

链表的链式存储结构使得它在处理这种循环报数和删除节点的问题上具有天然的优势。在链表中，我们可以很方便地通过修改指针来实现节点的删除操作，而不需要像数组那样移动大量元素。

链表的特性在这里得到了充分的体现。

链式存储

顺序存储是指将数据元素依次存放在一块连续的存储空间中，就像在一排连续的房间里依次存放物品一样。顺序存储的优点是可以通过下标直接访问元素，访问速度快，时间复杂度为 $O(1)$ 。但缺点也很明显，插入和删除元素时可能需要移动大量元素。比如在数组中间插入一个元素，需要将插入位置之后的所有元素都向后移动一位，时间复杂度为 $O(n)$ 。

而**链式存储**是通过节点之间的指针连接来组织数据，每个节点可以存放在内存的任意位置，就像用绳子将分散在各处的物品串起来一样。链表中的节点通过指针相互关联，形成一个链式结构。链式存储的优点是插入和删除元素比较方便，只需要修改指针的指向即可，时间复杂度为 $O(1)$ （前提是已经知道要操作的节点位置）。缺点是不能随机访问元素，要访问链表中的某个节点，必须从链表的头节点开始，依次遍历链表，时间复杂度为 $O(n)$ 。

节点结构

链表每个节点包含**数据域**和**指针域**，数据域用于存储实际的数据，可以是任意类型的数据，比如整数、字符、结构体等，如题可以存储人的编号。指针域用于存储指向下一个节点的指针，通过这个指针可以将各个节点连接起来，形成链表。

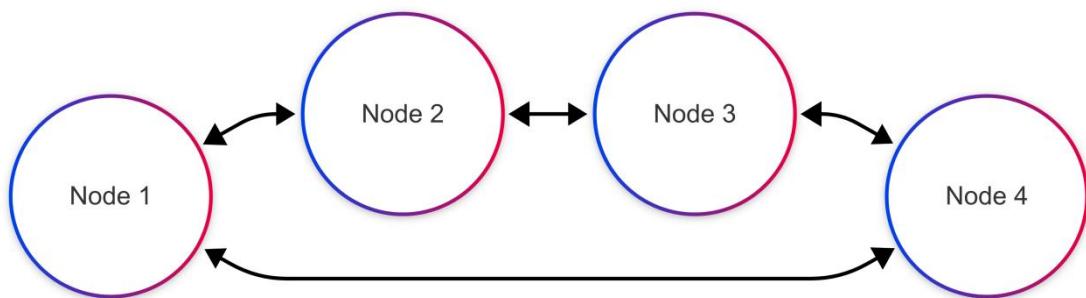


图1.1.1 指针域示意图

节点结构体是实现循环链表的要点，如示意图，循环由节点和指针构成的。我们正在实现的是双向链表，因此两个节点之间相互持有指向对方的指针。循环链表之所以叫循环，是因为最后一个节点与第一个节点连在了一起。通过使用循环双向链表，我们可以方便的模拟出题目要求的环形的报数。

下面是循环链表节点结构体定义的部分代码。

```
1 struct node
2 {
3     int val, nxt, pre;
4     //数据域-（双向）指针域
5 };
6
7 const int N = 120;
8 node t[N];
9 //使用数组写法便于调试，也有更好的常数优势（连续内存对cache友好）
```

图1.1.2 循环链表节点结构体定义的部分代码

首先，`struct node` 定义了一个节点结构体。这个结构体包含三个成员变量：数据域`val`,指针域`nxt`和`pre`，其中`nxt`指针指向链表中的下一个节点，`pre`指针指向链表中的前一个节点。这种双向指针的设计使得在链表中进行遍历和操作更加灵活，例如可以方便地向前或向后遍历链表，在删除节点时也更容易处理指针的更新。

其次，`const int N = 120`定义了一个常量 N，这个常量用于指定数组 t 的大小。
`node t[N];` 声明了一个类型为 node 的数组 t。

时间复杂度对比

在链表中删除一个节点的时间复杂度为 $O(1)$ ，而在数组中删除一个元素可能需要 $O(n)$ 的时间复杂度，因此链表在处理约瑟夫环问题时效率更高。由此我们可以对比出顺序存储和链式存储的时间复杂度差别。

操作	顺序存储（数组）	链式存储（链表）
插入（已知位置）	$O(n)$	$O(1)$
删除（已知位置）	$O(n)$	$O(1)$

图1.1.3 两种存储方式的时间复杂度对比

1.1.3 算法实现分步解析

步骤1：理解问题

已知约瑟夫环的问题是：n个人围成一圈，从第1个人开始报数，每报到m的人出列，直到所有人出列。这个过程要求按出列顺序输出每个人的编号。刚刚我们假设n=5，m=3时，出列顺序应为3→1→5→2→4。

通过上述思考问题动手操作，我们会发现利用已有的数组知识来解决这个问题时，需要删除元素，这个过程需要移动大量数据，大大增加了时间复杂度（ $O(n)$ ），这个时候需要一种支持高效删除的数据结构。

步骤2：数据结构选择

为了高效处理环形结构，避免时间复杂度以及空间复杂度过高，我们采用一种新的数据结构——链表（如下定义）。

```

4 struct node
5 {
6     int val, nxt, pre;
7     // 数据域-（双向）指针域
8 };
9
10 const int N = 120;
11 node t[N]; // 使用数组写法便于调试，也有更好的常数优势（连续内存对cache友好）
12 int cnt = 1;

```

图1.1.4 链表定义的代码

以下是每个节点的定义以及作用：

val：存储当前人的编号

nxt：指向下一个人的数组下标

pre：指向上一个人的数组下标

在这个过程我们用数组来模拟链表，这样做的好处是内存连续，访问速度快，且无需动态分配内存。

步骤3：初始化链表

代码如图1.1.5所示：

```

19     int n, m;
20     cin >> n >> m;
21     for (int i = 1; i <= n; i++)
22     {
23         t[i] = {i, i + 1, i - 1};
24     }
25     t[1].pre = n;
26     t[n].nxt = 1; // 形成循环链表

```

图1.1.5 初始化链表的代码

假设n=5， 初始化过程如下：

此时链表结构如下（箭头表示nxt方向）：

$1 \leftrightarrow 2 \leftrightarrow 3 \leftrightarrow 4 \leftrightarrow 5 \rightarrow 1$ （循环）

步骤4：模拟报数过程

代码如1.1.6图所示：

```

28     while (n--)
29     {
30         for (int i = 1; i <= m; i++)
31         {
32             cnt = t[cnt].nxt;
33         }
34         // 此时cnt指向的是第m + 1名同学
35         cnt = t[cnt].pre;
36         cout << t[cnt].val << ' ';
37         int front = t[cnt].pre, rear = t[cnt].nxt;
38         t[front].nxt = rear;
39         t[rear].pre = front;
40
41         // 下一个报数的同学是第m + 1个
42         cnt = t[cnt].nxt;
43     }

```

图1.1.6 模拟报数的代码

其中，变量cnt表示当前报数的起点。每次循环将执行以下操作：

·子步骤4.1：找到第m个人

通过移动m次cnt，此时cnt指向第m+1个人。例如：

初始cnt=1，m=3时：

第1次移动：cnt=2（报数1）

第2次移动：cnt=3（报数2）

第3次移动：cnt=4（报数3，此时cnt指向第4人）

此时，实际要出列的是第3人（即cnt的前驱）。

·子步骤4.2：删除当前节点

以n=5，m=3为例：

第一次循环：删除3，链表变为 $1 \leftrightarrow 2 \leftrightarrow 4 \leftrightarrow 5 \rightarrow 1$ ，下一轮从4开始。

第二次循环：移动3次（4→5→1→2），删除1，链表变为 $2 \leftrightarrow 4 \leftrightarrow 5 \rightarrow 2$ ，下一轮从2开始。

依此类推，直到所有人出列。

步骤5：完整代码展示

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 struct node
5 {
6     int val, nxt, pre;
7     // 数据域-（双向）指针域
8 };
9
10 const int N = 120;
11 node t[N]; // 使用数组写法便于调试，也有更好的常数优势（连续内存对cache友好）
12 int cnt = 1;
13
14 int main()
15 {
16     ios::sync_with_stdio(false);
17     cin.tie(nullptr);
18
19     int n, m;
20     cin >> n >> m;
21     for (int i = 1; i ≤ n; i++)
22     {
23         t[i] = {i, i + 1, i - 1};
24     }
25     t[1].pre = n;
26     t[n].nxt = 1; // 形成循环链表
27
28     while (n--)
29     {
30         for (int i = 1; i ≤ m; i++)
31         {
32             cnt = t[cnt].nxt;
33         }
34         // 此时cnt指向的是第m + 1名同学
35         cnt = t[cnt].pre;
36         cout << t[cnt].val << ' ';
37         int front = t[cnt].pre, rear = t[cnt].nxt;
38         t[front].nxt = rear;
39         t[rear].pre = front;
40
41         // 下一个报数的同学是第m + 1个
42         cnt = t[cnt].nxt;
43     }
44
45     return 0;
46 }

```

图1.1.7 完整代码



链表基础探索：单向链表（B3631）

实现一个数据结构，维护一张表（最初只有一个元素1）。需要支持下面的操作，其中x和y都是1到106范围内的正整数，且保证任何时间表中所有数字均不相同，操作数量不多于105：

1. 将元素y插入到x后面；
2. 询问x后面的元素是什么。如果x是最后一个元素，则输出0；
3. 从表中删除元素x后面的那个元素，不改变其他元素的先后顺序。

输入格式：第一行一个整数q表示操作次数。接下来q行，每行表示一次操作，操作具体见题目描述。

输出格式：对于每个操作2，输出一个数字，用换行隔开。

1.1.4 深入讨论

时间复杂度

(1) 访问元素

· 链表中的元素是通过指针依次访问的，因此访问特定元素需要从头节点开始遍历，时间复杂度是 $O(n)$ 。

(2) 插入元素

· 如果已知要插入位置的前一个节点，则插入操作只需要调整几个指针，时间复杂度是 $O(1)$ 。

· 如果不知道具体位置，则需要先遍历找到插入点，时间复杂度为 $O(n)$ 。

(3) 删除元素

· 如果已知要删除的节点，则删除操作只需要调整几个指针，时间复杂度是 $O(1)$ 。

· 如果不知道具体位置，则需要先遍历找到删除点，时间复杂度为 $O(n)$ 。

空间复杂度

链表的空间复杂度是 $O(n)$ ，其中 n 是链表中节点的数量。

每个节点除了存储数据外，还需要存储指向下一个节点的指针（以及可能的前一个节点的指针，如果是双向链表）。

因此，链表相对于数组会有额外的指针存储空间开销。

易错点提示

下表中展示了我们在处理链表问题时容易出现的一些错误及相应的解决方法：

错误类型	后果	解决方法
空链表直接操作	程序崩溃	操作前检查 <code>if (head == nullptr)</code>
头尾节点更新遗漏	链表断裂或内存泄漏	传递引用或二级指针，记录前驱节点
单节点处理不当	野指针或逻辑错误	删除后显式置空头指针
双向链表指针不一致	链表结构破坏	插入/删除时同步更新 <code>prev</code> 和 <code>next</code>

图1.1.8 常见错误分析表

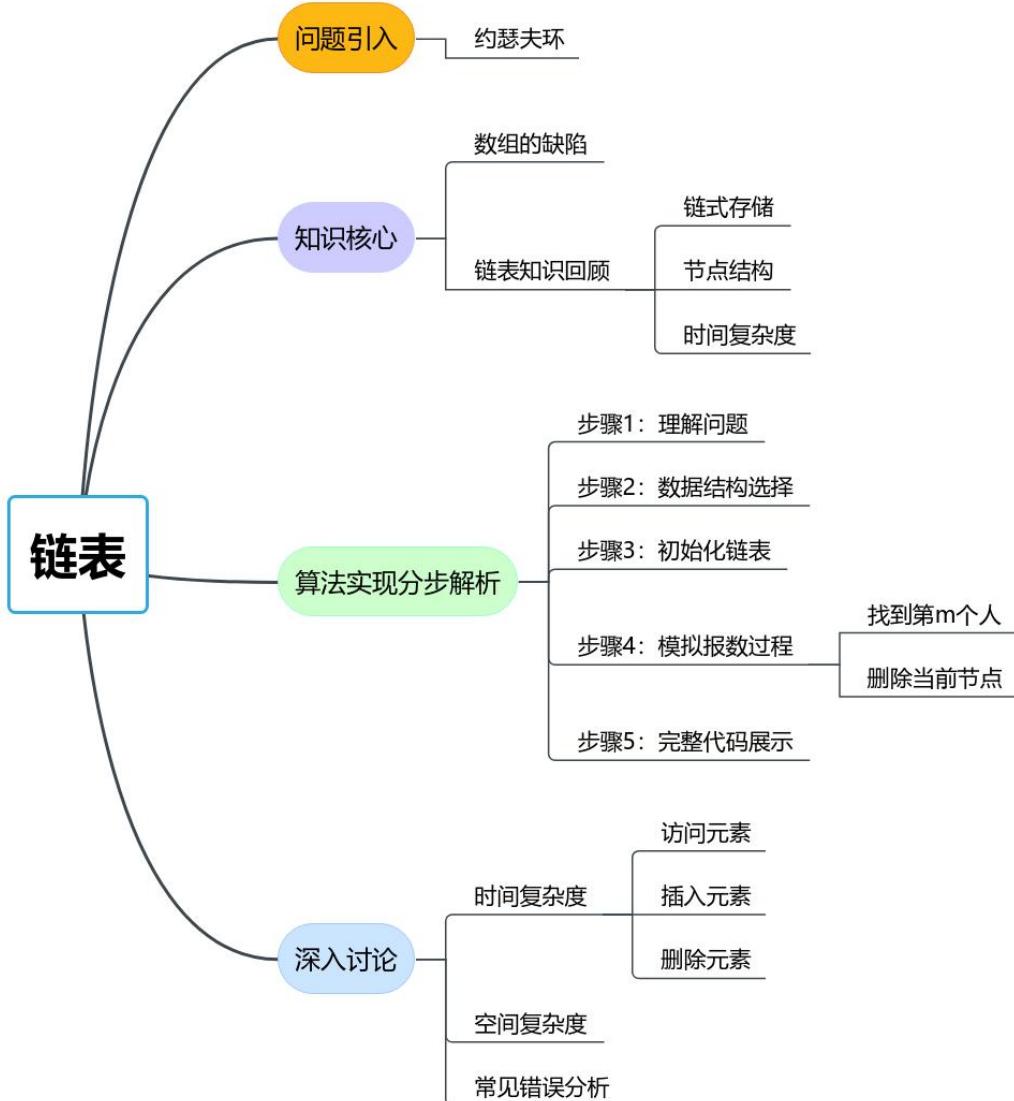


链表高级应用：邻值查找 (P1046)

给定一个长度为 n 的序列 A ， A 中的数各不相同。对于 A 中的每一个数 A_i ，求： $\min_{1 \leq j < i} |A_i - A_j|$ 以及令上式取到最小值的 j （记为 P_i ）。若最小值点不唯一，则选择使 A_j 较小的那个。

输入格式：第一行输入整数 n ，代表序列长度。第二行输入 n 个整数 $A_1 \sim A_n$ ，代表序列的具体数值，数值之间用空格隔开。

输出格式：输出共 $n-1$ 行，每行输出两个整数，数值之间用空格隔开。分别表示当 i 取 $2 \sim n$ 时，对应的 $\min_{1 \leq j < i} |A_i - A_j|$ 和 P_i 的值。



练习提升

1. 编写程序，利用随机函数，生成 10 个 $0 \sim 10$ 的随机整数存储在数组中，并判断其中是否有数字“5”，若有，则输出它在数组中的下标（如有多个，也一并输出）；否则，输出“NO DATA”。
2. 编写程序，使用二维数组构造出以下的杨辉三角形（要求输出n行， $n > 9$ ）。
杨辉三角形是南宋数学家杨辉所著的《详解九章算术》一书中用三角形解释二项式系数的乘方规律，是二项式系数在三角形中的一种几何排列。

```
1  
1 1  
1 2 1  
1 3 3 1  
1 4 6 4 1  
1 5 10 10 5 1  
1 6 15 20 15 6 1  
⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮
```

1.2 栈——更好的对称匹配

本节学习目标

- ◆ 理解栈的基本概念与特性，能够对比栈与数组、链表在插入、删除操作上的性能差异。
- ◆ 掌握栈的基本操作，并能够用数组或链表模拟实现栈结构。
- ◆ 理解栈操作的时间复杂度，并能够解释其优势。
- ◆ 识别适合栈解决的问题场景，独立设计基于栈的解决方案。

1.2.1 问题引入：闭合判别

问题描述

给定一个仅包含 () 和 [] 的字符串，判断括号是否合法闭合。

具体实例

若输入：(0[])(0[])，则输出：YES；若输入：(0[])(0[])，则输出：NO。

1.2.2 问题拆解

合法闭合

在解决这个问题之前，我们先来回顾一下合法闭合这个关键概念。

在编程中，特别是在处理表达式求值和括号匹配问题时，合法闭合是至关重要的。合法闭合指的是按照正确的顺序关闭或配对括号的过程。例如：在算术表达式中，每个左括号“（”都需要有一个对应的右括号“）”来闭合它，反之亦然。合法闭合确保了表达式的结构正确性和逻辑一致性。

那开动咱们聪明的小脑筋，如果有多个括号，我们该如何快速判断括号的嵌套关系呢？这里提示一个关键点：在括号匹配过程中，最后出现的左括号需要最优先匹配。很矛盾是不是？那根据这个关键点是否能想出相关的数据结构呢？

栈

不错，想必大家已经想到了，这可以通过使用“栈”这种数据结构来实现。栈是一种“后进先出”（LIFO）的线性数据结构。

每当遇到一个左括号时，就将其推入栈中；每当遇到一个右括号时，就检查栈顶是否有对应的左括号。如果有，就将左括号从栈中弹出，表示一对括号成功匹配。这也代表着最后出现的左括号就是最后一个被推入栈中的（此时是栈顶，好比图1.2.1盘子堆的最顶上的盘子，在堆放盘子时它是最后一个放上去的），同时也是第一个从栈中弹出的（从一摞盘子中拿出一个盘子，都是从最顶上第一个开始拿）。



图1.2.1 一摞盘子

但“后进先出”（LIFO）也只是栈的一个主要特征，我们一起来看看栈的其他基本操作吧。

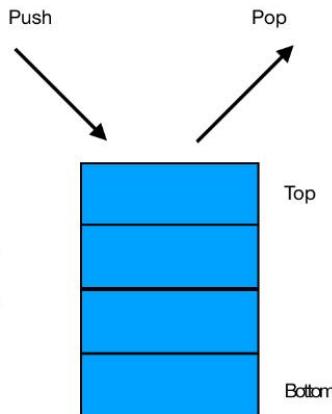


图1.2.2 后进先出操作示意图

栈的基本操作

栈的基本操作主要包括四点，如下表：

操作	作用
push ()	将元素压入栈
pop ()	弹出栈顶元素
top ()	获取栈顶元素
bottom ()	获取栈底元素
size ()	统计栈中元素数量

1.2.3 用数组模拟栈

在之前我们回顾了栈的基本用法，知道栈是一种后进先出（Last In First Out, LIFO）的数据结构。接下来，我们来看看如何用数组来模拟栈的操作。

首先，我们需要定义一个数组和一个指针来模拟栈。数组用来存储栈中的元素，指针用来记录栈顶的位置。在这里，N 定义了我们模拟栈的最大容量为 1000，st 数组就是我们的栈空间，而 ptr 是栈顶指针，初始值为 0，表示栈为空，它指向的是下一个可以插入元素的位置。

```
const int N = 1000;
int st[N]; // 栈空间
int ptr = 0; // 栈顶指针（指向下一个空位）
```

图1.2.3 定义数组和指针

接下来，我们实现栈的几个基本操作：

- **压栈操作（push）**：将元素放入栈中。

在 push 函数中，我们先将 ptr 指针自增 1，让它指向下一个空位，然后把要压入栈的元素 x 存储在 st[ptr] 的位置上。这样就完成了一次压栈操作。

```
void push(int x) { st[ptr] = x; }
```

图1.2.4 压栈操作

- 弹栈操作（pop）：从栈中取出元素。

pop 函数用于弹出栈顶元素。我们先检查 ptr 是否大于 0，如果大于 0，说明栈中至少有一个元素，此时将 ptr 指针减 1，相当于把栈顶元素移除了。如果 ptr 等于 0，说明栈为空，不进行任何操作。

```
void pop() { if (ptr > 0) ptr--; }
```

图1.2.5 弹栈操作

- 获取栈顶元素（top）：查看栈顶的元素但不弹出它。

top 函数直接返回 st[ptr] 的值，也就是栈顶元素的值。不过要注意，在调用这个函数之前最好先确保栈不为空，否则返回的结果是没有意义的。

```
int top() { return st[ptr]; }
```

图1.2.6 获取栈顶元素

- 获取栈的大小（size）：了解栈中当前元素的数量。

size 函数很简单，直接返回 ptr 的值，因为 ptr 记录的是栈中元素的个数（同时也是下一个空位的位置）。

```
int size() { return ptr; }
```

图1.2.7 获取栈的大小

1.2.4 算法实现分步解析

步骤1：理解问题

已知问题要求是给定一个仅由 ()[] 组成的字符串，判断其是否合法。这个时候我们思考什么情况下这个字符串是合法，即合法的条件。首先字符串要满足

闭合性——每个右括号必须有对应的左括号。其次要满足顺序性——括号必须按正确顺序闭合（例如 "(D]" 不合法，因为），闭合时最近的左括号是 []。

■ 示例：

合法: "O[]"、"(())"。

非法: "(D]"（顺序错）、"(O"（未闭合）、"]O"（右括号未匹配左括号）。

步骤2：数据结构选择

通过问题分析，我们发现这个问题的核心需求是需要快速找到当前右括号对应的左括号，且必须是最新未闭合的左括号。这个问题解决过程符合栈的特性，即后进先出（Last In First Out）。后进入栈的左括号会先被匹配，完美契合括号的闭合顺序。

接下来我们来对比其他结构：

数组/链表：需要额外指针记录最近左括号位置，不如栈直接。

哈希表：无法直接处理顺序问题。

综上所述，栈是解决此类“对称匹配问题”的最优数据结构。

步骤3：栈的初始化

代码如图1.2.8所示：

```
4 const int N = 1000;
5 char s[N];
6 int ptr = 0;
7
8 void push(char x)
9 {
10    s[ptr] = x;
11 }
12
13 void pop()
14 {
15    ptr--;
16 }
17
18 char top()
19 {
20    return s[ptr];
21 }
22
23 int size()
24 {
25    return ptr;
26 }
```

图1.2.8 初始化栈的代码

用字符数组 `s[N]` 模拟栈，`ptr` 是栈顶指针（初始为0，表示空栈）。

■ 关键操作：

`push(x)`: 将 `x` 压入栈顶（先让 `ptr` 加1，再存入 `x`）。

`pop()`: 栈顶指针减1（相当于删除栈顶元素）。

`top()`: 返回当前栈顶元素（即 `s[ptr]`）。

`size()`: 返回栈中元素个数（即 `ptr` 的值）。

步骤4：遍历字符串处理括号

代码如图所示：

```

28 int main()
29 {
30     ios::sync_with_stdio(false);
31     cin.tie(nullptr);
32     string str;
33     cin >> str;
34     bool flag = true;
35     for (auto c : str)
36     {
37         if (c == '(' || c == '[')
38         {
39             push(c);
40         }
41         else if (c == ')' && top() == '(')
42         {
43             pop();
44         }
45         else if (c == ']' && top() == '[')
46         {
47             pop();
48         }
49         else
50         {
51             flag = false;
52             break;
53         }
54     }
55     if (size())
56     {
57         flag = false;
58     }
59 }

```

图1.2.9 遍历字符串处理括号代码

遍历字符串的每个字符 c:

- 子步骤4.1: 处理左括号 ((或 [)

直接压入栈中，等待后续匹配。

例子：输入 "([", 栈会依次存入 "(" 和 "["。

- 子步骤4.2: 处理右括号 () 或])

检查栈是否为空：如果栈为空（比如输入 "]")，说明没有对应的左括号，标记非法（flag = false）。

检查栈顶是否匹配：

如果 c 是) 且栈顶是 (，弹出栈顶。

如果 c 是] 且栈顶是 [，弹出栈顶。

例子：输入 "()", 遇到) 时栈顶是 (，匹配后栈变空。

不匹配的情况：如果栈顶不匹配（比如 c 是] 但栈顶是 ()），标记非法。

步骤5：完整代码展示

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 const int N = 1000;
5 char s[N];
6 int ptr = 0;
7
8 void push(char x)
9 {
10     s[ptr] = x;
11 }
12
13 void pop()
14 {
15     ptr--;
16 }
17
18 char top()
19 {
20     return s[ptr];
21 }
22
23 int size()
24 {
25     return ptr;
26 }
27
28 int main()
29 {
30     ios::sync_with_stdio(false);
31     cin.tie(nullptr);
32     string str;
33     cin >> str;
34     bool flag = true;
35     for (auto c : str)
36     {
37         if (c == '(' || c == '[')
38         {
39             push(c);
40         }
41         else if (c == ')' && top() == '(')
42         {
43             pop();
44         }
45         else if (c == ']' && top() == '[')
46         {
47             pop();
48         }
49         else
50         {
51             flag = false;
52             break;
53         }
54     }
55     if (size())
56     {
57         cout << flag;
58     }
59 }
```

图1.2.10 完整代码

■ 注意：

遍历后栈必须为空！（`if (size() != 0) flag = false;`）如果栈不为空，强制标记为非法。

1.2.5 扩展应用：栈的典型场景

我们成功运用数组模拟栈的方法，巧妙解决了括号匹配问题，对栈的基本操作与应用有了扎实掌握。接下来我们将目光投向一类非常经典且更具挑战性的问题——利用栈求表达式的值。

请思考一下如何求出表达式 $3 * (2 + 5)$ 的值。

表达式求值的关键在于**处理运算符的优先级**。对于加、减、乘、除以及括号，乘除运算优先级高于加减，而括号内的运算又优先于括号外。为了准确计算，我们需要按正确顺序处理运算符和操作数，因此我们会用到两个栈，一个用于存储操作数，另一个用于存储运算符。

(1) 扫描表达式

先从左到右逐个扫描表达式中的字符。

(2) 处理操作数

当遇到数字时，将其解析为完整的数值，然后压入操作数栈。例如，遇到3，就把3压入操作数栈；遇到2和5时，也依次压入。

(3) 处理运算符

当遇到左括号“（”时，直接将其压入运算符栈。这是因为左括号标志着一个子表达式的开始，后续遇到的运算符和操作数会在这个子表达式内处理。

当遇到运算符（如+、-、*、/）时，需要与运算符栈顶的运算符比较优先级。若当前运算符优先级高于栈顶运算符，就将当前运算符压入运算符栈；若当前运算符优先级低于或等于栈顶运算符，就从操作数栈中弹出两个操作数，从运算符栈中弹出一个运算符进行计算，然后把计算结果压回操作数栈，接着继续比较当前运算符与新的栈顶运算符的优先级，重复上述过程，直到当前运算符可以压入运算符栈。

(4) 最终计算

扫描完整个表达式后，运算符栈中可能还残留运算符。此时，依次从操作数栈弹出两个操作数，从运算符栈弹出一个运算符进行计算，将结果压回操作数栈，直到运算符栈为空。此时，操作数栈中剩下的唯一元素就是整个表达式的计算结果。

以下是完整代码：

```
1 #include <iostream>
2 #include <stack>
3 #include <string>
4 #include <cctype>
5
6 // 获取运算符优先级
7 int precedence(char op) {
8     if (op == '+' || op == '-') return 1;
9     if (op == '*' || op == '/') return 2;
10    return 0;
11}
12 // 执行运算
13 int applyOp(int a, int b, char op) {
14    switch (op) {
15        case '+': return a + b;
16        case '-': return a - b;
17        case '*': return a * b;
18        case '/':
19            if (b == 0) throw std::runtime_error("Division by zero");
20            return a / b;
21    }
22    return 0;
23}
24 // 表达式求值
25 int evaluate(const std::string& tokens) {
26    std::stack<int> values;
27    std::stack<char> ops;
28
29    for (size_t i = 0; i < tokens.length(); i++) {
30        if (tokens[i] == ' ') continue;
31
32        if (isdigit(tokens[i])) {
33            int val = 0;
34            while (i < tokens.length() && isdigit(tokens[i])) {
35                val = (val * 10) + (tokens[i] - '0');
36                i++;
37            }
38            i--;
39            values.push(val);
40        } else if (tokens[i] == '(') {
41            ops.push(tokens[i]);
42        }
43    }
44
45    while (!ops.empty()) {
46        int b = values.top();
47        values.pop();
48        int a = values.top();
49        values.pop();
50        char op = ops.top();
51        ops.pop();
52        int result = applyOp(a, b, op);
53        values.push(result);
54    }
55
56    return values.top();
57}
```

图1.2.11 求表达式的值代码（上）

```

42     } else if (tokens[i] == ')') {
43         while (!ops.empty() && ops.top() != '(') {
44             int val2 = values.top();
45             values.pop();
46             int val1 = values.top();
47             values.pop();
48             char op = ops.top();
49             ops.pop();
50             values.push(applyOp(val1, val2, op));
51         }
52         if (!ops.empty()) ops.pop();
53     } else {
54         while (!ops.empty() && precedence(ops.top()) >= precedence(tokens[i])) {
55             int val2 = values.top();
56             values.pop();
57             int val1 = values.top();
58             values.pop();
59             char op = ops.top();
60             ops.pop();
61             values.push(applyOp(val1, val2, op));
62         }
63         ops.push(tokens[i]);
64     }
65 }
66
67 while (!ops.empty()) {
68     int val2 = values.top();
69     values.pop();
70
71 return 0;

```

图1.2.12 求表达式的值代码（下）

1.2.6 单调栈

问题描述

n 个人正在排队进入一个音乐会。人们等得很无聊，于是他们开始转来转去，想在队伍里寻找自己的熟人。

队列中任意两个人 a 和 b ，如果他们是相邻或他们之间没有人比 a 或 b 高，那么他们是可以互相看得见的。

写一个程序计算出有多少对人可以互相看见。

对于全部的测试点，保证 $1 \leq$ 每个人的高度 $< 2^{31}$ ， $1 \leq n \leq 5 \times 10^5$ 。

问题分析

a. 暴力做法：对于每对人检查中间是否有比他们高的，时间复杂度为 $O(n^2)$ ，显然无法通过本题。

b. 优化思路：想象所有人在你面前站成一排，观察：对于队伍中的一个人*i*，他能看到的人有什么规律？



图1.2.13 排队示意图

他们的身高具有**单调性**。并且，这种单调性可以在枚举*i*的过程中维护。

比方说，你在第*i*个人处已经得到了前*i*-1个人构成的“能看到的人序列”，这个序列的长度就是第*i*个人向左看能看到的人数。

现在你怎样计算第*i*+1个人向左看能看到的人数呢？

这不难，只要把第*i*个人加入序列维护出答案就好。如果第*i*个人比序列最右边的人高，就把序列最右的人排除，因为他之后会被第*i*人挡住。请注意被排除的人很可能不止一个人。

每个人都只会被枚举一次，最多被加入序列一次，最多被排除一次，因此这种做法的时间复杂度是 $O(n)$ 。

怎么实现呢？你又注意到我们需要这个序列的大小，还需要对这个序列的最右端做操作。现在你联想到了什么数据结构？栈。

代码实现

■ 重点：

- (1) 使用栈未判空导致的越界
- (2) 边界条件：多个身高相同的人

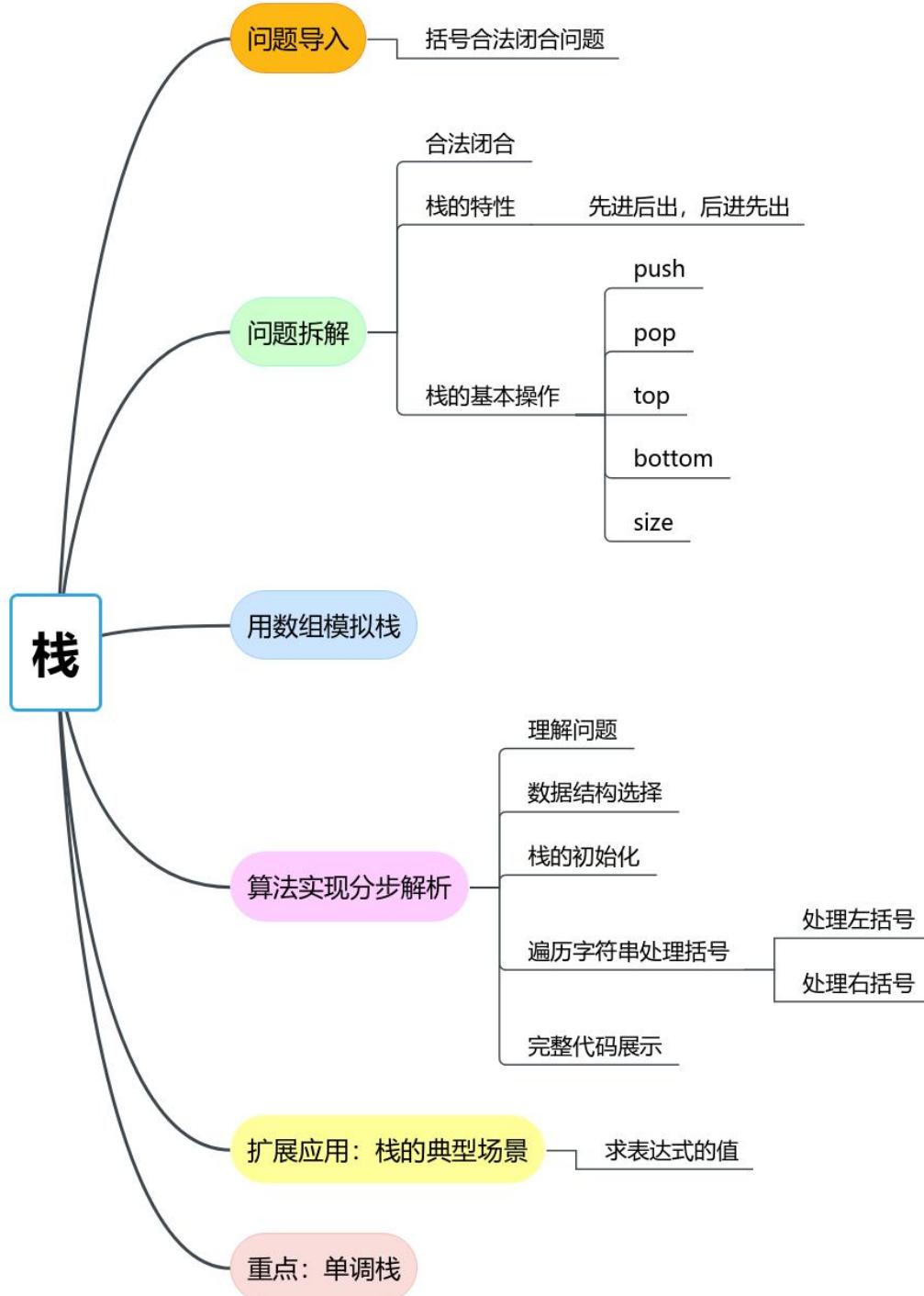
```

1 #include <bits/stdc++.h>
2 using namespace std;
3 using ll = long long;
4 const int N = 5e5 + 10;
5 using person = pair<ll, int>;
6 person st[N];
7 int ptr = 0;
8
9 void push(person x)
10 {
11     st[ptr] = x;
12 }
13 void pop()
14 {
15     if (ptr > 0)
16         ptr--;
17 }
18 auto top()
19 {
20     return st[ptr];
21 }
22 int size()
23 {
24     return ptr;
25 }
26
27 int main()
28 {
29     ios::sync_with_stdio(false);
30     cin.tie(nullptr);
31
32     ll n, ans = 0;
33     cin >> n;
34     for (int i = 1; i ≤ n; i++)
35     {
36         ll height;
37         cin >> height;
38         person cnt = {height, 1};
39
40         while (size() && top().first ≤ height)
41         {
42             if (top().first == height)
43                 cnt.second += top().second;
44             ans += top().second;
45             pop();
46         }
47
48         if (size())
49             ans++;
50
51         push(cnt);
52     }
53     cout << ans;
54     return 0;
55 }
56

```

图1.2.14 单调栈完整代码

这样的技巧就被称为单调栈。





练习提升

1. 编写程序，利用栈将从键盘输入的十进制整数转换为十六进制数，并输出。
2. 一般情况下，求自然数 n 的阶乘 $n!$ ，需要知道 $(n-1)!$ ，依此类推，思考这一解题思路中是否用到栈的原理，小组讨论，说明理由。

1.3 队列——有序的先进先出

本节学习目标

- ◆ 理解队列的基本概念与特性。
- ◆ 掌握队列的基本操作，并能够解决队列相关问题。
- ◆ 理解队列操作的时间复杂度，并能够解释其优势。
- ◆ 识别适合队列解决的问题场景，独立设计基于队列的解决方案。
- ◆ 能够通过队列的应用，提升计算机思维和问题解决能力。

1.3.1 情境引入：滑动窗口

问题描述

有一个长为 n 的序列 a ，以及一个大小为 k 的窗口。现在这个从左边开始向右滑动，每次滑动一个单位，求出每次滑动后窗口中的最大值和最小值。

具体实例

对于序列 $[1,3,-1,-3,5,3,6,7]$ 以及 $k=3$ ，有如下过程：

窗口位置	最小值	最大值
$[1 \quad 3 \quad -1] \quad -3 \quad 5 \quad 3 \quad 6 \quad 7$	-1	3
1 $[3 \quad -1 \quad -3] \quad 5 \quad 3 \quad 6 \quad 7$	-3	3
1 3 $[-1 \quad -3 \quad 5] \quad 3 \quad 6 \quad 7$	-3	5
1 3 -1 $[-3 \quad 5 \quad 3] \quad 6 \quad 7$	-3	5
1 3 -1 -3 $[5 \quad 3 \quad 6] \quad 7$	3	6
1 3 -1 -3 5 $[3 \quad 6 \quad 7]$	3	7

图1.3.1 滑动窗口操作示意图

输入格式：

输入一共有两行，第一行有两个正整数 n, k 。第二行 n 个整数，表示序列
a。例如：

```
8 3  
1 3 -1 -3 5 3 6 7
```

图1.3.2 滑动窗口输入示例

输出格式：

输出共两行，第一行为每次窗口滑动的最小值，第二行为每次窗口滑动的最大值。例如：

```
-1 -3 -3 -3 3 3  
3 3 5 5 6 7
```

图1.3.3 滑动窗口输出示例

1.3.2 队列基础概念与实现

核心特性FIFO

队列（queue）是一种具有「先进入队列的元素一定先出队列」性质的表。
由于该性质，队列通常也被称为先进先出（first in first out）表，简称 **FIFO**。

为了更好地理解队列的特性，我们可以把队列类比成地铁安检通道。在安检通道里，先进入通道的乘客会先完成安检，后进入的乘客则需要在后面排队等待，依次接受安检，这和队列中的元素进出顺序是完全一致的。



图1.3.4 队列示意图

STL基本操作

在实际编写代码时，我们不需要自己去实现队列这种数据结构，C++ 标准模板库（STL）已经为我们提供了功能完善的 `std::queue`，我们只需要学会如何使用它就好。接下来，让我们看看 `std::queue` 常用的基本操作：

- `queue<int> q;`

此语句定义了一个名为 `q` 的队列对象，这个队列能够存储 `int` 类型的元素。这里的 `queue` 实际上是 `std::queue` 的简写，不过要保证在代码开头含 `<queue>` 头文件。

- `q.push();`

`push` 是 `std::queue` 的成员函数，其作用是把一个元素添加到队列的尾部。例如 `q.push (1)` 是将整数 1 加入到队列 `q` 里。

- `q.pop();`

`pop` 函数用于移除队列头部的元素。不过需要注意的是，在调用 `pop` 之前，要先检查队列是否为空，因为对空队列调用 `pop` 会引发未定义行为。

- `q.front();`

`front` 函数会返回队列头部元素的引用。同样，在调用 `front` 之前，也要确保队列不为空，不然会出现未定义行为。

- `q.size();`

`size` 函数返回队列中元素的数量。

- `q.empty();`

`empty` 函数用来判断队列是否为空。若队列为空，返回 `true`；反之，返回 `false`。

STL双端队列

除了 `std::queue`，STL 中还有一种与之相关的数据结构——**双端队列(deque)**。双端队列 `deque` 支持在 $O(1)$ 时间复杂度内进行首尾插入和删除操作。它和 `queue` 最大的区别在于，`deque` 既可以从队首进（出）队，又可以从队尾进（出）队，使用起来更加灵活。下面是 `deque` 的一些常见操作示例：

- `deque<int> dq;`

这行代码定义了一个名为 dq 的双端队列对象，该队列可以存储 int 类型的元素。这里的 deque 实际上是 std::deque 的简写，不过要确保在代码的开头包含 <deque> 头文件。

- **dq.push_back();**

push_back 是 std::deque 的成员函数，其作用是将一个元素添加到双端队列的尾部。例如 dq.push_back(2)，把整数 2 加入到双端队列 dq 的尾部。

- **dq.pop_front();**

pop_front 函数用于移除双端队列头部的元素。同样，在调用 pop_front 之前，需要先检查双端队列是否为空，因为对空的双端队列调用 pop_front 会导致未定义行为。

1.3.3 滑动窗口实现

步骤1：理解问题

已知问题要求：给定一个长度为 n 的序列 a 和一个大小为 k 的窗口，窗口从最左端开始向右滑动，每次滑动一个单位，输出每次窗口滑动后的最小值和最大值。

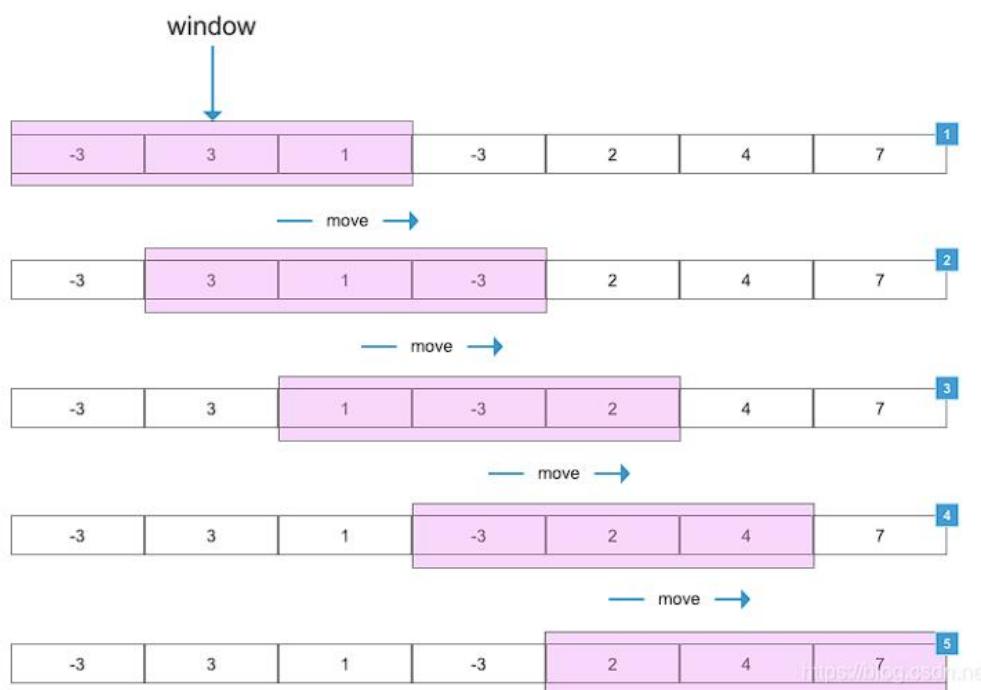


图 1.3.5 滑动窗口

示例分析：

输入序列为 [1,3,-1,-3,5,3,6,7]，窗口大小 k=3，输出最小值序列为 -1,-3,-3, -3,3,3，最大值序列为 3,3,5,5,6,7。

关键约束：需在 $O(n)$ 时间复杂度内完成，避免暴力枚举的低效操作。

步骤2：数据结构选择

通过问题分析，发现核心需求是快速维护窗口范围，并高效获取极值。传统队列无法直接满足需求，因此引入**单调队列**：

- 单调递增队列：维护窗口最小值，队首始终为当前窗口最小值的索引。
- 单调递减队列：维护窗口最大值，队首始终为当前窗口最大值的索引。

对比其他结构：

- 数组/链表：遍历窗口内元素求极值的时间复杂度为 $O(k)$ ，无法满足高效性。
- 优先队列（堆）：虽然能快速获取极值，但无法高效删除滑动过程中移出窗口的元素。

同时，单调队列通过动态维护队列的单调性，可在 $O(1)$ 时间内获取极值，并保证操作整体时间复杂度为 $O(n)$ ，是最优选择。

步骤3：队列初始化与维护规则

关键操作逻辑（以最小值为例）：

- 初始化双端队列 `deque<int> dq`，存储元素索引。
- 遍历序列：对每个元素 $a[i]$ ，执行以下操作：
 - 移除过期元素：若队首索引 $dq.front() \leq i-k$ 且 $dq.back() \leq i-k$ ，说明该元素已不在窗口内，弹出队首。
 - 维护单调性：从队尾开始，若当前元素 $a[i] \leq a[dq.back()]$ ，则弹出队尾元素，直到队列恢复单调递增。
 - 插入当前索引：将 i 加入队尾。

- 记录结果：当 $i \geq k-1$ 时，队首元素即为当前窗口的最小值。

最大值队列的逻辑与最小值类似，只需将比较条件改为 $a[i] \geq a[dq.back()]$ 。

代码如1.3.2图所示：

```

vector<int> min_ans, max_ans;
deque<int> dq;

// 处理最小值队列（递增）
for (int i = 0; i < n; ++i) {
    while (!dq.empty() && dq.front() <= i - k) {
        dq.pop_front();
    }
    while (!dq.empty() && a[i] <= a[dq.back()]) {
        dq.pop_back();
    }
    dq.push_back(i);
    if (i >= k - 1) {
        min_ans.push_back(a[dq.front()]);
    }
}

```

图1.3.5 处理最小值

步骤4：完整代码展示

```

1 #include <iostream>
2 #include <vector>
3 #include <deque>
4 using namespace std;
5
6 int main() {
7     ios::sync_with_stdio(false);
8     cin.tie(nullptr);
9
10    int n, k;
11    cin >> n >> k;
12    vector<int> a(n);
13    for (int i = 0; i < n; ++i) {
14        cin >> a[i];
15    }
16
17    vector<int> min_ans, max_ans;
18    deque<int> dq;
19
20    // 处理最小值队列（递增）
21    for (int i = 0; i < n; ++i) {
22        while (!dq.empty() && dq.front() <= i - k) {
23            dq.pop_front();
24        }
25        while (!dq.empty() && a[i] <= a[dq.back()]) {
26            dq.pop_back();
27        }
28        dq.push_back(i);
29        if (i >= k - 1) {
30            min_ans.push_back(a[dq.front()]);
31        }
32    }

```

图1.3.6 完整代码（上）

```

35     // 处理最大值队列(递减)
36     for (int i = 0; i < n; ++i) {
37         while (!dq.empty() && dq.front() ≤ i - k) {
38             dq.pop_front();
39         }
40         while (!dq.empty() && a[i] ≥ a[dq.back()]) {
41             dq.pop_back();
42         }
43         dq.push_back(i);
44         if (i ≥ k - 1) {
45             max_ans.push_back(a[dq.front()]);
46         }
47     }
48
49     // 输出结果
50     for (int i = 0; i < min_ans.size(); ++i) {
51         if (i) cout << " ";
52         cout << min_ans[i];
53     }
54     cout << '\n';
55
56     for (int i = 0; i < max_ans.size(); ++i) {
57         if (i) cout << " ";
58         cout << max_ans[i];
59     }
60     cout << '\n';
61
62
63     return 0;
64 }
```

图1.3.6 完整代码（下）

易错点提示

- **错误原因：**直接调用`q.front()`，当队列为空时，这会引发问题。
- **解决方法：**在调用`front()`或`pop()`之前，先用`empty()`方法检查队列是否为空。

```

1 // 错误示例：未检查队列空直接访问
2 int val = q.front();
3 // q为空时导致运行时错误
4 // 正确写法
5 if(!q.empty())
6 {
7     val = q.front();
8     q.pop();
9 }
```

图1.3.7 队列越界问题

1.3.5 总结与扩展应用：BFS广度优先搜索

题目描述

马的遍历（P1443）：

有一个 $n \times m$ 的棋盘，在某个点 (x, y) 上有一个马，要求你计算出马到达棋盘上任意一个点最少要走几步。对于全部的测试点，保证 $1 \leq x \leq n \leq 400$, $1 \leq y \leq m \leq 400$ 。

输入输出

• 输入格式

输入只有一行四个整数，分别为 n, m, x, y 。

3 3 1 1

图1.3.8 输入范例

• 输出格式

一个 $n \times m$ 的矩阵，代表马到达某个点最少要走几步（不能到达则输出 -1）

0	3	2
3	-1	1
2	1	4

图1.3.9 输出范例

核心思路

（1）问题建模

将棋盘视为一个无权图，每个格子是图的节点。马从起点出发，每次按“日”字形移动（8种方向），求到达所有节点的最短路径。

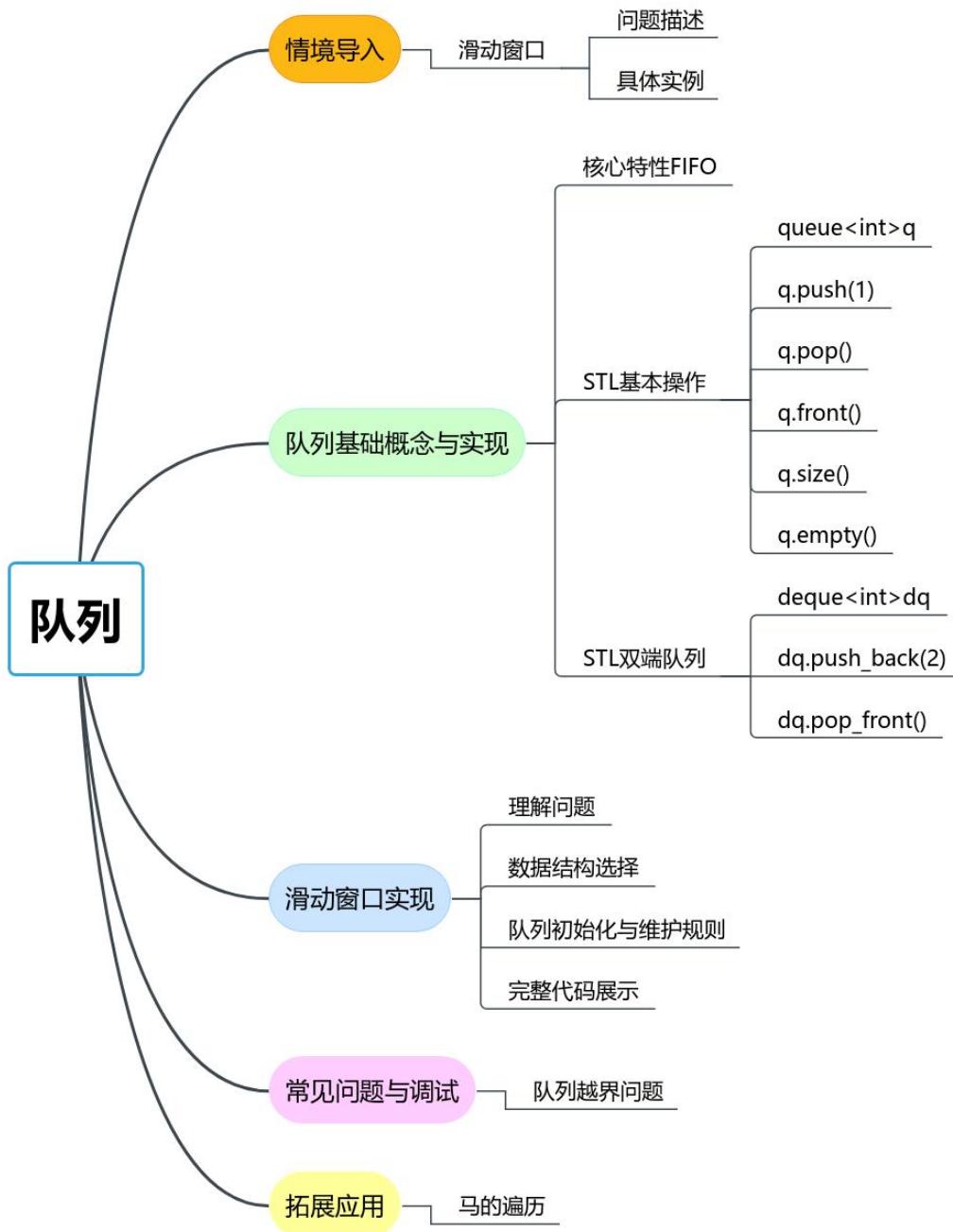
（2）BFS 的适用性

BFS 按层遍历，首次访问到某节点时所用的步数即为最短步数，天然适合解决单源最短路径问题。

核心代码展示

```
1  queue<pt> q;
2  q.push({x, y});
3  int cnt = 0;
4  auto check = [n, m, &dis](pt a) {
5      if (a.x > n || a.x < 1 || a.y > m || a.y < 1)
6          return false;
7      if (dis[a.x][a.y] == inf)
8          return true;
9      return false;
10 };
11 dis[x][y] = 0;
12 while (!q.empty())
13 {
14     auto now = q.front();
15     q.pop();
16     auto [a, b] = now;
17     cnt = dis[a][b];
18     for (int j = 0; j < 8; j++)
19         if (check({a + X[j], b + Y[j]}))
20             q.push({a + X[j], b + Y[j]}), dis[a + X[j]][b + Y[j]] = cnt + 1;
21 }
```

图1.3.10 马的遍历问题核心代码展示

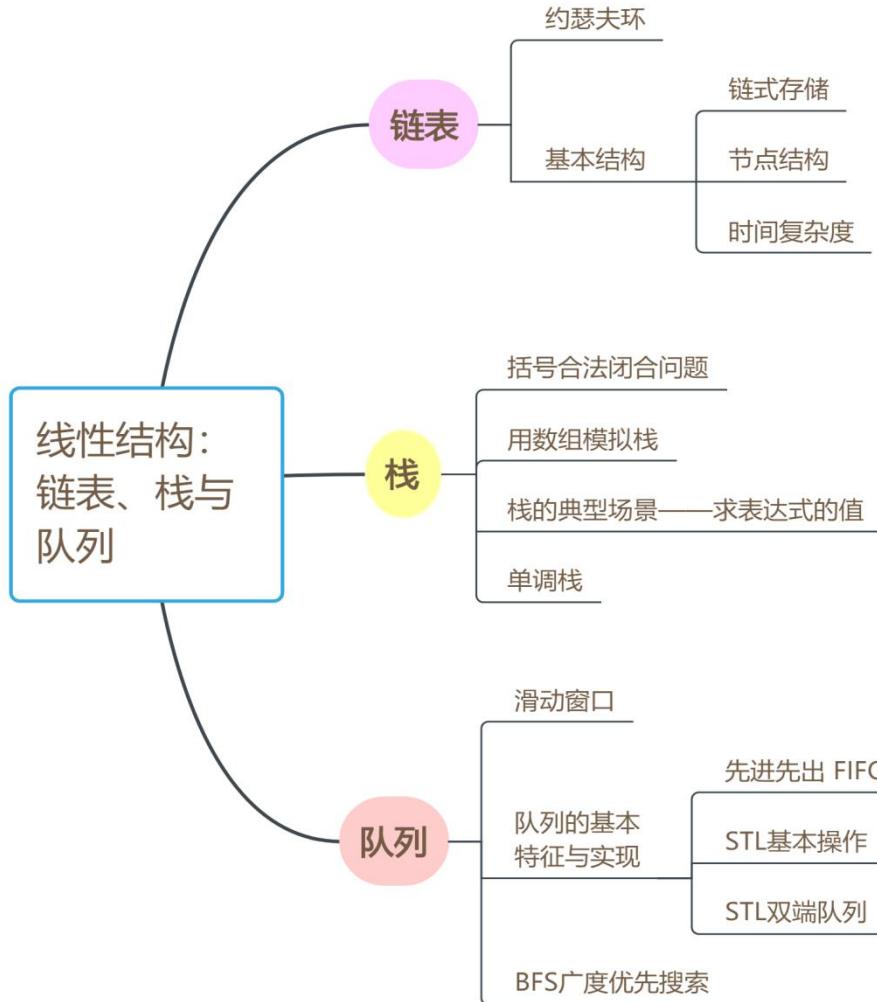




练习提升

1. 假设用长度为 11 的数组作为顺序队列，初始为空。分别绘出做完下列操作后的队列示意图。若有元素不能入队，试说明理由。操作序列如下：
 - (1) A, B, C, D, E 依次入队； (2) A, B 依次出队；
 - (3) F, G, H, I, J 依次入队； (4) O, P, Q, R 依次入队。
2. 利用两个顺序栈 S1 和 S2 模拟一个队列。利用栈的基本操作，实现队列的入队和出队操作，并说出基本思路。

1. 下图展示了本章的核心概念与关键能力，请同学们对照图中的内容进行总结。



2. 根据自己的掌握情况填写下表。

学习内容	掌握程度
链表的基本结构	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
链表的应用	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
用数组模拟栈	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
栈的典型场景应用	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
队列的基本特征与实现	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
队列的应用	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解

小结

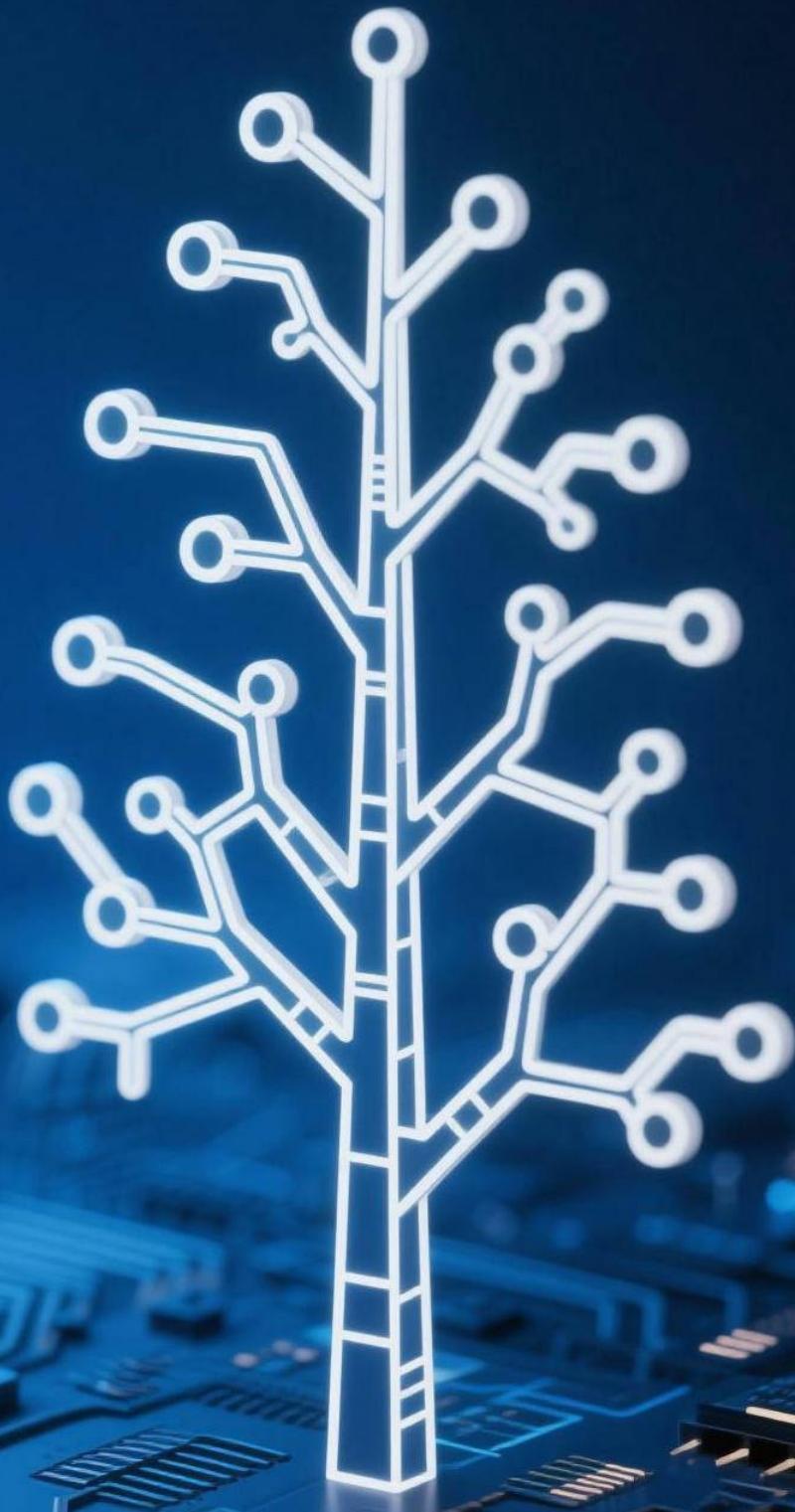
在本章中，我们围绕链表、栈和队列这三种基础且核心的数据结构展开了深入学习。通过经典竞赛真题的实战演练，我们不仅掌握了这些数据结构的基本用法，更学会从实际问题出发，依据数据特点和操作需求，精准选择合适的数据结构。这不仅提升了我们解决问题的效率，更是理解高级数据结构与复杂算法的重要铺垫。希望大家能将本章所学融会贯通，为后续 CSP-J 竞赛征程夯实基础，在面对各类难题时，灵活运用这些数据结构，展现扎实的编程功底与卓越的算法思维。

第二章

树：树与二叉树、字典树

在一片茂密的森林中，每一棵树都从根部向上生长，分出枝干，再蔓延出细小的叶片，形成层次分明的结构。这种自然的层级之美，正是计算机科学中树形结构的灵感来源——无论是家族谱系中代代相传的血脉，还是文件系统中目录与子目录的嵌套关系，树都能以简洁的规则描述复杂的层次逻辑。

在编程的世界里，树是一种非线性数据结构，它摆脱了线性结构的单一顺序，用分支和层级为数据赋予更丰富的表达力。从数据库索引的快速检索，到网络路由的高效寻址；从决策树的智能分类，到语法树的代码解析，树的身影无处不在。本章将带您从基础出发，深入理解树的数学定义与核心术语，掌握其存储方法与遍历技巧，并通过经典问题剖析树在算法中的巧妙应用。



2.1 树的基本概念

本节学习目标

- ◆ 理解树的数学定义与基本性质，能够区分树与图的区别。
- ◆ 掌握树的基本术语（如根、叶、深度、高度等）及其实际意义。
- ◆ 识别特殊树类型（链、菊花、二叉树）的结构特点。
- ◆ 掌握邻接表存储树的方法，并能用代码实现树的遍历。

2.1.1 问题引入：家谱关系建模

问题描述

某家族需要构建家谱图，要求表示成员间的父子关系，并支持快速查询祖先、后代及兄弟关系。如何用数据结构高效表示这种层级关系？

具体实例

假设家族成员关系如下：

- (1) A 是 B 和 C 的父亲
- (2) B 是 D 和 E 的父亲
- (3) C 是 F 的父亲

2.1.2 知识核心

树的定义

树是满足以下任一条件的连通无向图：

- 有 n 个结点和 $n-1$ 条边，且无环。
- 任意两结点间有且仅有一条简单路径。
- 所有边均为桥（删除任意边会导致图不连通）。

如图2.1.1，一棵包含 6 个结点的树，边连接方式为层级结构。

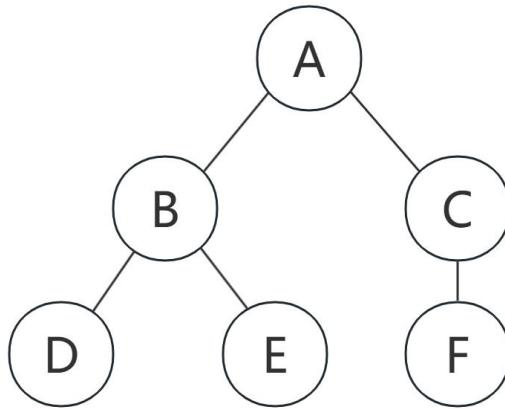


图2.1.1 树结构示意图

树的基本概念

◆ 父结点 (parent node)

在树结构中，若结点 A 直接连接到结点 B 的上层，则称 A 是 B 的父结点。除根结点外，每个结点有且仅有一个父结点。根结点没有父结点。

◆ 祖先 (ancestor)

若存在从结点 B 到结点 A 的路径，则 A 是 B 的祖先。根结点的祖先集合为空。

◆ 子结点 (child node)

若结点 B 是结点 A 的直接下层结点，则 B 是 A 的子结点。一个结点可以有零个或多个子结点，具体数量由树的类型决定。

◆ 结点的深度 (depth)

从根结点到该结点的路径上经过的边的数量。

◆ 树的高度 (height)

从某结点到其最远叶子结点的最长路径上的边数。

◆ 兄弟 (**sibling**)

具有相同父结点的结点互为兄弟结点。

◆ 后代 (**descendant**)

若存在从结点 A 到结点 B 的路径(通过连续向下访问子结点), 则 B 是 A 的后代。

◆ 子树 (**subtree**)

以树中某结点为根结点, 包含其所有后代结点及连接的边构成的子结构。

◆ 森林 (**forest**)

由零个或多个互不相交的树组成的集合。

◆ 无根树的叶结点 (**leaf node**)

在无根树中, 度数为 1 的结点称为叶结点。

◆ 有根树的叶结点 (**leaf node**)

在有根树中, 没有子结点的结点称为叶结点。

特殊的树

(1) 链 (**chain/path graph**) : 链是一种退化的树结构, 所有结点按线性顺序排列, 每个结点(除首尾结点外)仅有一个父结点和一个子结点。

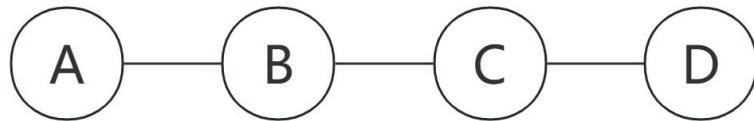


图2.1.2 链

(2) 菊花/星星 (**star**) : 由一个中心根结点和多个直接连接的叶结点组成, 无中间层级的树结构。

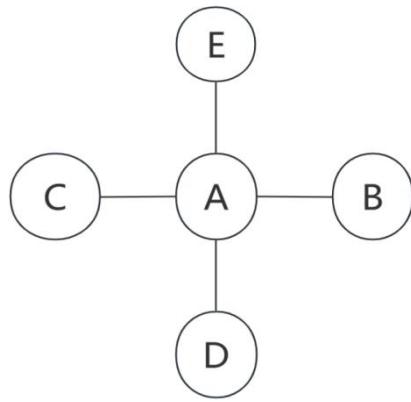


图2.1.3 菊花/星星

(3) 有根二叉树 (rooted binary tree) : 有根二叉树是一种每个结点最多有两个子结点的有根树。

2.1.3 利用邻接表存储树

理解问题

邻接表由顶点数组（或哈希表）和链表（或动态数组）组成。每个顶点对应一个独立的链表，链表中存储与该顶点直接相连的所有相邻顶点。

已知要利用邻接表存储树，我们需要为每个结点开辟一个线性列表，记录所有与之相连的结点。

示例展示

```
#include <iostream>
#include <vector>
using namespace std;

const int N = 100010;
vector<int> adj[N];

int main() {
    int n, m; /
    cin >> n >> m;
    for (int i = 0; i < m; ++i) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    return 0;
}
```

图2.1.4 利用邻接表存储树

2.1.4 利用DFS遍历树

概念介绍

DFS（Depth-First Search，深度优先搜索）是一种用于遍历或搜索树、图等数据结构的算法。其核心思想是尽可能深地探索分支路径，直到无法继续前进，再回溯到上一个分叉点尝试其他路径。

示例展示

```
void dfs (int cnt, int f) {
    for (int i : adj[cnt]) {
        if (i != f) {
            dfs(i, cnt);
        }
    }
}
```

图2.1.4 利用DFS遍历树



思考辨析

如果将示例程序中的 `if(i != f)` 语句删去会发生什么？

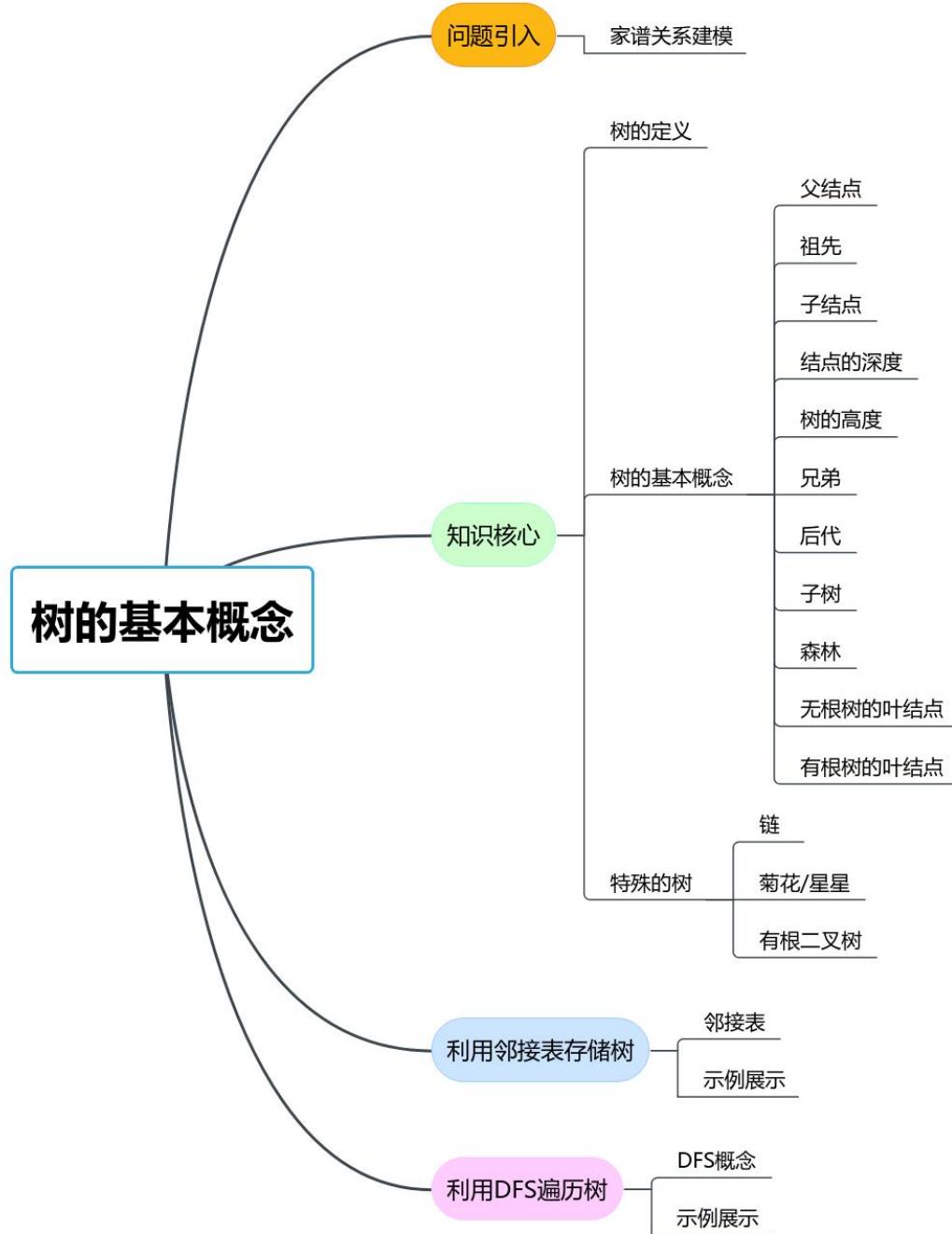
删除示例程序中的 `if(i != f)` 会导致无限递归和栈溢出。`if(i != f)` 是确保 DFS 正确遍历的关键条件，删除后程序将无法正常运行。

◆ 原因：

1. 防止回溯父节点：该条件的作用是避免当前节点 `i` 重新访问其父节点 `f`。在树或图的遍历中，父节点已被处理过，无需重复访问。
2. 避免循环：若删除此条件，当遍历到父节点时，程序会递归调用 `dfs(f, cnt)`，而 `f` 又会再次访问当前节点 `cnt`，形成 `cnt ↔ f` 的无限循环。
3. 栈溢出：递归深度无限制增加，最终因函数调用栈耗尽而崩溃。

◆ 示例：

假设节点 1 和 2 相连。调用 `dfs(1, -1)` 会访问 2，接着调用 `dfs(2, 1)`。若未跳过父节点，2 会再次访问 1，触发 `dfs(1, 2)`，1 又访问 2……无限循环。





一、判断题

1. 树中根节点没有父节点，其他节点有且仅有一个父节点。 ()
2. 叶子节点是指没有子节点的节点。 ()
3. 树中任意两个节点之间的边构成一条唯一的路径。 ()
4. 节点的“度”是指该节点的兄弟节点数量。 ()

二、填空题

1. 树中节点的子节点数量称为该节点的_____。
2. 树中从根节点到某一节点经过的边数称为该节点的_____。
3. 根节点的直接子节点称为其_____，根节点是这些子节点的_____。
4. 多个互不相交的树构成的集合称为_____。

三、选择题

1. 以下关于“树的高度”描述正确的是：
 - A) 根节点的高度为0
 - B) 树的高度等于最深叶子节点的深度
 - C) 树的高度等于节点总数
 - D) 树的高度等于边数
2. 节点A的父节点的父节点是节点B，则节点B是节点A的：
 - A) 兄弟节点
 - B) 祖先节点
 - C) 子节点
 - D) 后代节点
3. 以下属于树的基本术语的是：
 - A) 链表

B) 叶子

C) 哈希表

D) 栈

4. 树中同一父节点的子节点互为：

A) 祖先

B) 后代

C) 兄弟

D) 根

四、术语匹配题

将左侧术语与右侧定义连线：

1. 根节点 A. 树中节点的最大深度

2. 森林 B. 没有父节点的节点

3. 路径 C. 由边连接的节点序列

4. 树的高度 D. 多个互不相交的树

五、简答题

1. 定义解释

什么是树的“子树”？

2. 术语对比

“节点的深度”和“节点的高度”有何区别？

2.2 二叉树——倒置的树

本节学习目标

- ◆ 掌握二叉树的定义，包括空树和非空树的组成（根结点、左子树、右子树）。
- ◆ 熟悉关键术语：叶子结点、内部结点、结点的度、深度、高度，并能够通过示意图分析二叉树结构。
- ◆ 通过递归实现前序、中序、后序遍历，并能够根据表达式树写出对应的遍历结果。

2.2.1 二叉树的定义

形式化定义

二叉树（Binary Tree）是一种满足以下条件的树形数据结构：

- (1) 空树：不包含任何结点的二叉树。
- (2) 非空树：由以下三部分组成：
 - ① 根结点（Root）：唯一没有父结点的结点。
 - ② 左子树（Left Subtree）：根结点的左子结点及其后代构成的二叉树。
 - ③ 右子树（Right Subtree）：根结点的右子结点及其后代构成的二叉树。

每个结点最多有两个子结点，分别称为 **左子结点（Left Child）** 和 **右子结点（Right Child）**。子结点的顺序严格区分， $\text{左子结点} \neq \text{右子结点}$ ，不可交换位置。

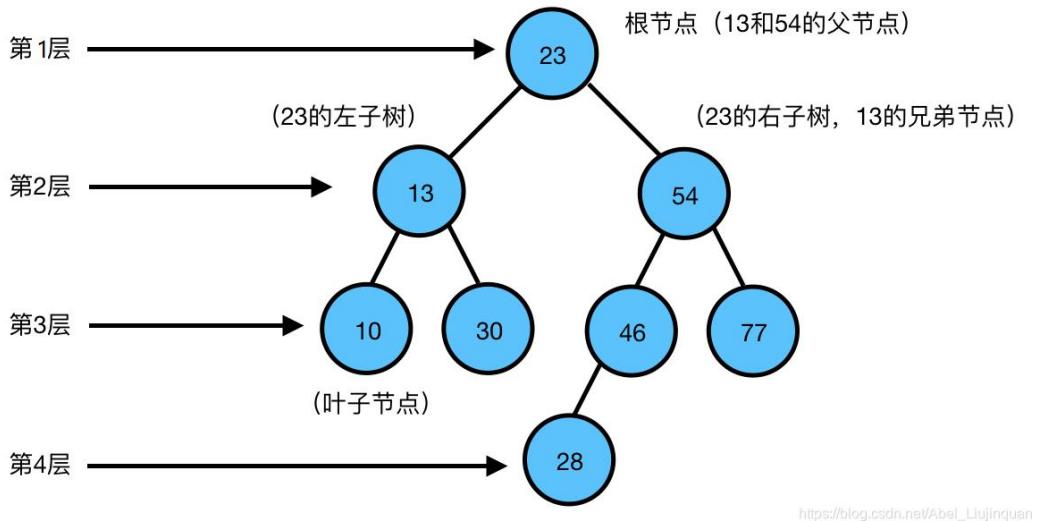


图2.2.1 二叉树结构示意图

关键术语

◆ 叶子结点 (Leaf Node)

没有子结点的结点（度为0）。

◆ 内部结点 (Internal Node)

至少有一个子结点的结点（度 ≥ 1 ）。

◆ 结点的度 (Degree)

一个结点拥有的子结点数量（二叉树中度为0、1或2）。

◆ 深度 (Depth)

根结点到该结点的路径长度（根结点深度为0或1，依定义约定）。

◆ 高度 (Height)

结点到叶子结点的最长路径长度（叶子结点高度为0或1，依定义约定）。

2.2.2 二叉树的性质

层、深度与结点数的关系

性质	公式	说明
第 i 层最多结点数	$2^{(i-1)}$	几何级数增长
深度 k 的二叉树最大结点数	$2^k - 1$	满二叉树时成立

图2.2.2 层、深度与结点数关系表

结点、度关系原理

设叶子结点数为 n_0 ，度为2的结点数为 n_2

恒等式： $n_0 = n_2 + 1$

证明思路：总边数 $B = n_1 + 2n_2 = n_0 + n_1 + n_2 - 1$



拓展提升：满二叉树

定义：深度为 k 的二叉树中，所有非叶子结点均有两个子结点，且所有叶子结点均位于同一层（第 k 层）。

性质：

① 结点总数： $2^k - 1$ （若根结点深度为1）

② 叶子结点数： 2^{k-1}

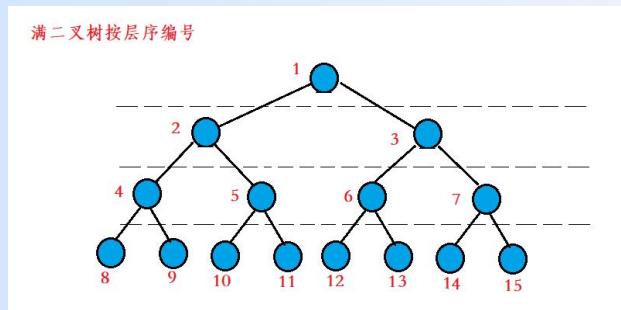


图2.2.3 满二叉树结构示意图

2.2.3 二叉树的存储

结构体数组法

结构体数组法是一种通过数组来静态存储二叉树结点及其子结点关系的实现方式。其核心思想是：

- ① 用数组下标表示结点的唯一标识（编号）。
- ② 通过两个独立数组（lchild[] 和 rchild[]）分别记录每个结点的左子结点和右子结点的下标。时也是第一个从栈中弹出的（从一摞盘子中拿出一个盘子，都是从最顶上第一个开始拿）。

```
1 int lchild[N], rchild[N]; // 下标表示结点编号
```

图2.2.4 结构体数组法代码示例

完全二叉树的数组存储

下标 i 的左子为 $2*i$ ，右子为 $2*i+1$

C++

```
1 #define ls(X) (x << 1)  
2 #define rs(X) (ls(X) + 1)
```

图2.2.5 完全二叉树的数组存储代码示例

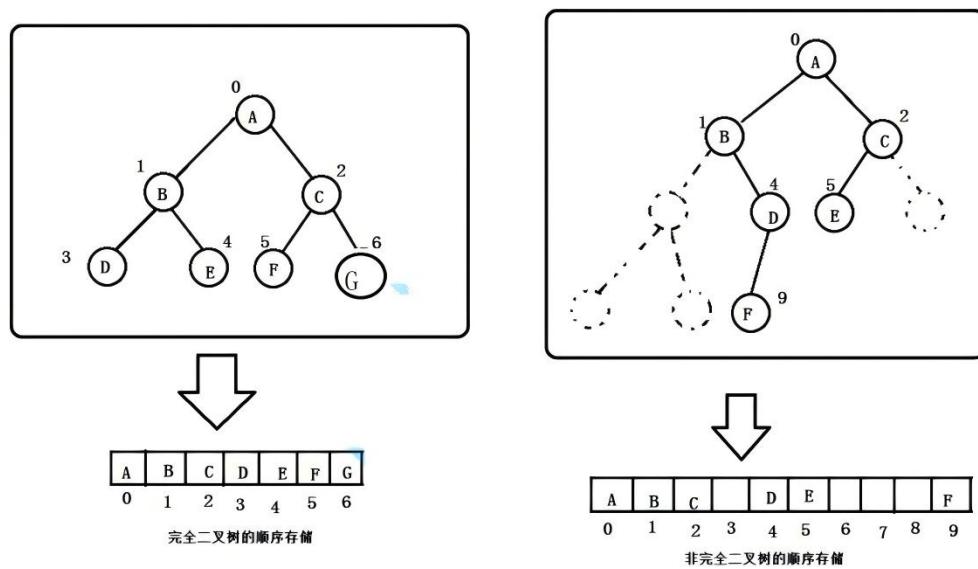


图2.2.6 数组存储与树形结构对应关系图

2.2.4 二叉树的遍历

递归遍历

- 前序遍历 (Pre-order Traversal) : 根节点 → 左子树 → 右子树
- 中序遍历 (In-order Traversal) : 左子树 → 根节点 → 右子树
- 后序遍历 (Post-order Traversal) : 左子树 → 右子树 → 根节点

```
1 void preorder(int u) {  
2     if (u == -1) return; // -1 表示空结点  
3     cout << u << " ";  
4     preorder(lchild[u]);  
5     preorder(rchild[u]);  
6 }
```

图2.2.7 递归遍历代码示例

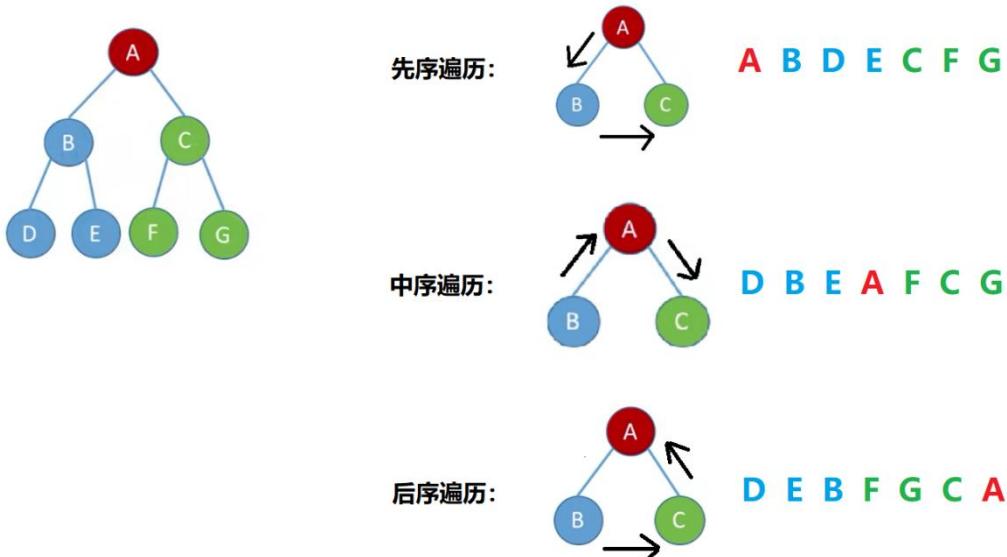


图2.2.8 三种遍历示意图

层次遍历

层次遍历 (Level Order Traversal) , 又称广度优先遍历 (BFS) , 是一种按树的层级逐层访问节点的算法。与递归实现的深度优先遍历 (前序/中序/后序) 不同, 层次遍历使用队列辅助实现, 确保先访问离根节点最近的节点。

- 核心思想

(1) 按层处理: 从根节点开始, 依次访问第1层、第2层……直到最后一层。

(2) 队列辅助：利用队列的先进先出（FIFO）特性，确保节点按层级顺序被访问。

- 实现步骤

(1) 初始化队列：将根节点加入队列。

(2) 循环处理队列：

节点出队 → 访问该节点 → 将子节点（先左后右）入队。

重复直到队列为空。

```
C++
```

```
1 queue<int> q;
2 q.push(root);
3 while (!q.empty()) {
4     int u = q.front(); q.pop();
5     cout << u << " ";
6     if (lchild[u] != -1) q.push(lchild[u]);
7     if (rchild[u] != -1) q.push(rchild[u]);
8 }
```

图2.2.9 层次遍历代码示例

遍历应用实例：表达式树

表达式树（Expression Tree）是一种用树形结构表示数学表达式的二叉树，能直观反映运算优先级和结合性，。以下是其核心要点：

1. 结构特点

叶子节点：存储操作数（数字或变量）。

内部节点：存储运算符（如 +, -, *, /）。

子树规则：每个运算符节点的左右子树分别表示其操作数。

2. 示例解析

以表达式 $3 + 4 * 5$ 为例：

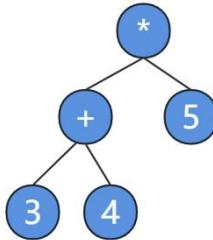


图2.2.10 表达式树示意图

前序对应前缀表达式，中序对应中缀表达式，后序对应后缀表达式。

遍历结果：

前序: * + 3 4 5 → 波兰表达式

中序: 3 + 4 * 5 → 中缀表达式（需加括号）

后序: 3 4 + 5 * → 逆波兰表达式



思考辨析

已知前序和中序序列，能否唯一确定二叉树？

已知前序遍历和中序遍历序列时，在节点值唯一的前提下，可以唯一确定一棵二叉树，这是二叉树遍历序列的重要性质。

◆ 示例演示：

假设前序序列为 1, 2, 4, 5, 3，中序序列为 4, 2, 5, 1, 3：

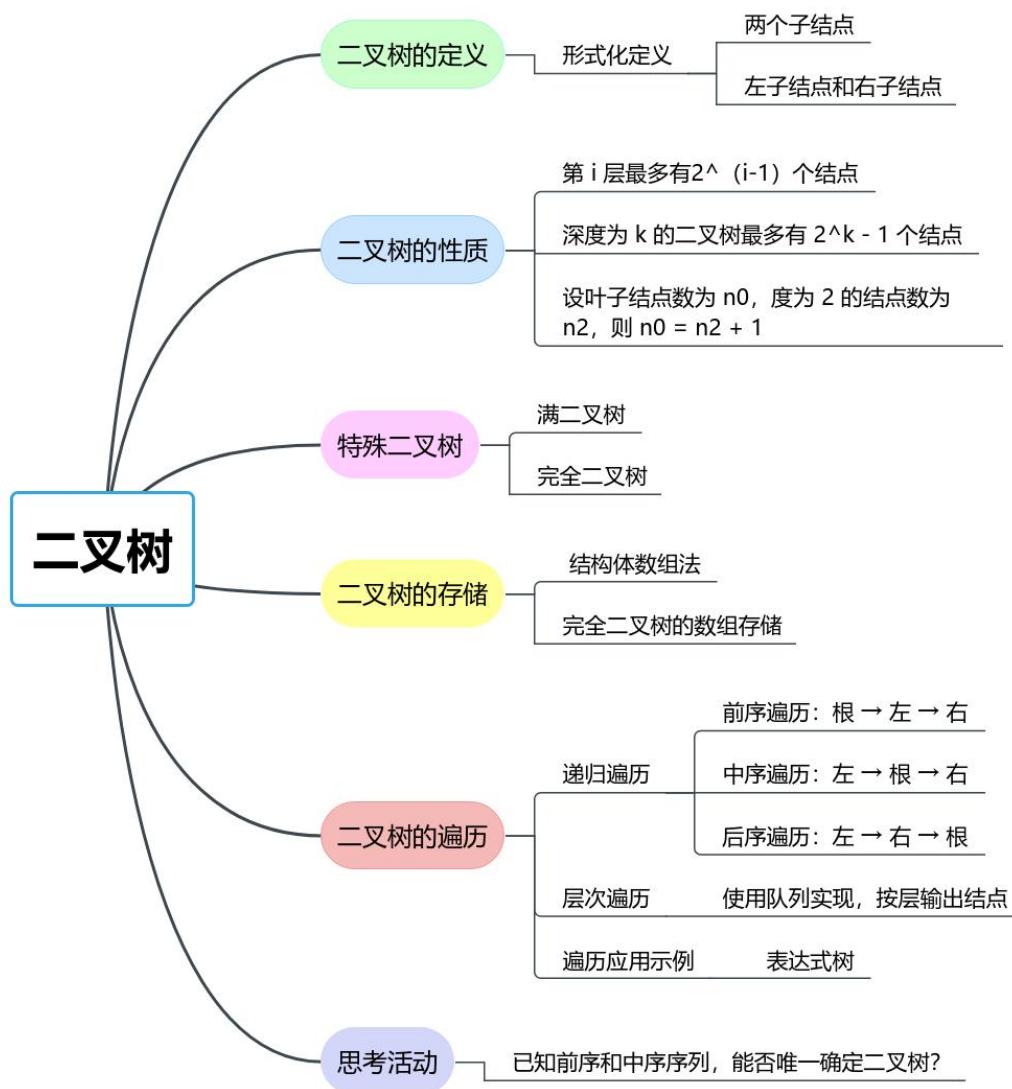
1. 前序首元素 1 是根节点。
2. 中序中 1 左侧 4, 2, 5 是左子树，右侧 3 是右子树。
3. 左子树有 3 个节点，因此前序中 2, 4, 5 是左子树的前序序列。
4. 对左子树递归：前序 2 是左子树根，中序 4, 2, 5 中 2 左侧 4 是左子树，右侧 5 是右子树。
5. 最终重建的树结构唯一。

◆ 特殊情况：

- ① 节点值重复：若存在重复值（如两个节点值均为 2），可能导致不同树结构产生相同的前序和中序序列。
- ② 单节点或空树：此时序列唯一，树结构显然唯一。



知识图谱



练习提升

- 根据定义, 找出图 2.2.11 中的二叉树。

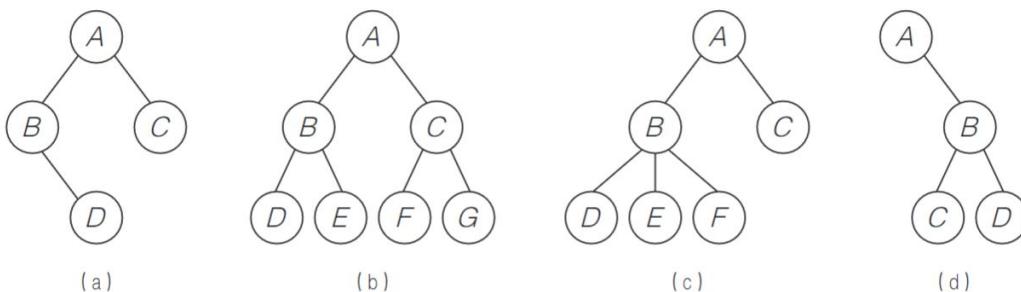


图2.2.11 找出符合条件的二叉树

2. 某二叉树有 3 个节点，两位同学就这棵二叉树的形态产生了争执，一位同学认为该二叉树的形态如图 2.2.12 (a) , 另一位同学则认为该二叉树的形态如图 2.2.12 (b) 。你赞同哪位同学的看法？还有其他的形态吗？

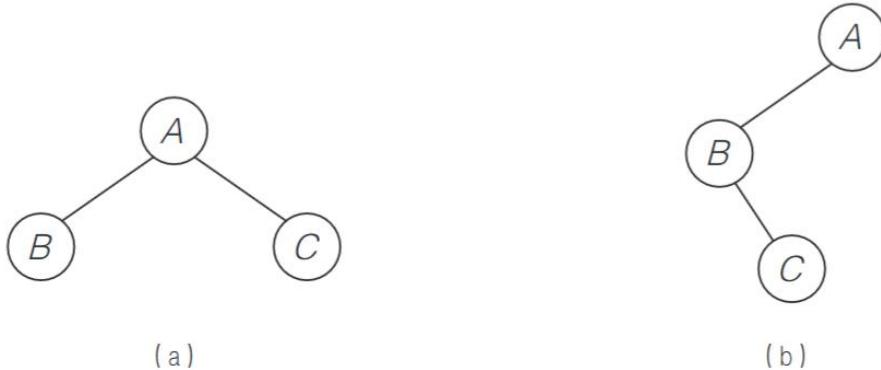


图2.2.12 3个节点的二叉树形态

2.3 树的使用案例——字典树

本节学习目标

- ◆ 理解字典树的基本概念与核心特征。
- ◆ 掌握字典树的基本操作，并能够解决字典树相关问题。
- ◆ 理解字典树的性质，并能够解释其与二叉树对比的优势。
- ◆ 识别适合字典树解决的问题场景，独立设计基于字典树的解决方案。
- ◆ 能够通过字典树的应用，提升计算机思维和问题解决能力。

2.3.1 字典树的定义

字典树（Trie树）

在计算机的世界里，当需要处理大量字符串时，高效的数据结构就显得尤为重要。今天，我们将一起认识一种专门用于存储和检索字符串的数据结构——字典树，也叫Trie树。

字典树是一种利用边来存储字符，从而实现多个字符串存储的数据结构。你可以把它想象成一棵多叉树，树的每一条路径，都代表着一个字符串。在字典树里，每个节点本身并不存储字符，而是通过节点之间的边所代表的字符，串联构成字符串。（如图2.3.1中a、b等字符存储在节点之间的边中）

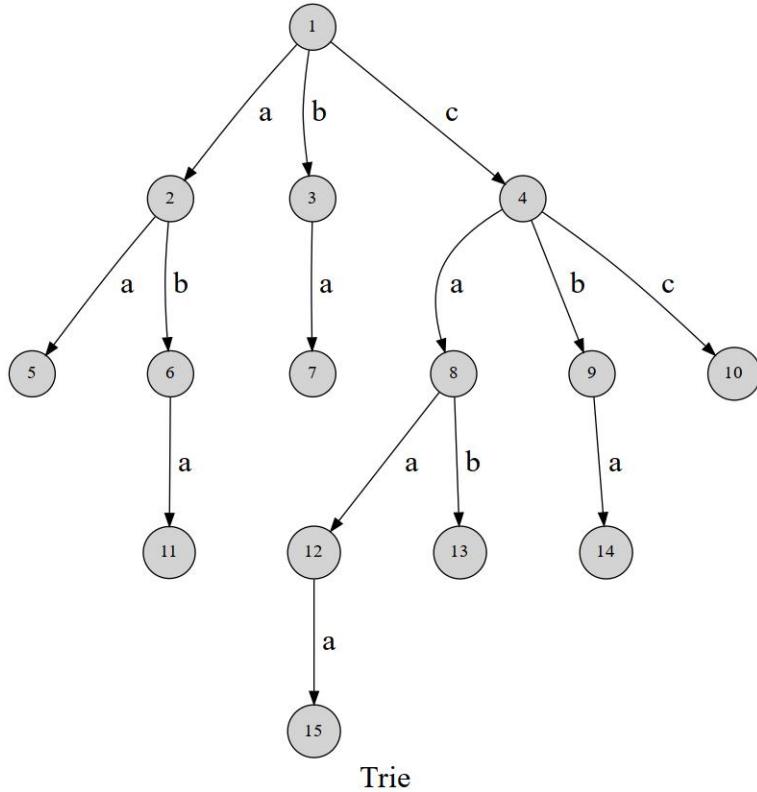


图2.3.1 字典树的结构示意图

具体实例

举个例子，当我们要在字典树中存储“cat” “car” “cart” 这几个单词时，从树的根节点出发，第一条边代表字母“c”，通过“c”边到达的节点，再延伸出代表“a”的边，沿着“a”边继续前进，“t”边和“r”边分别对应“cat”和“car”。对于“cart”，则是在“car”路径基础上，从代表“r”的节点继续延伸出“t”边。

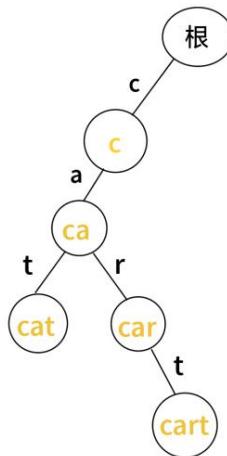


图2.3.2 字典树实例

由以上的图例我们可以看出，字典树有两个核心特征：多叉树结构、节点不存储字符。大家也可以发动脑筋进行续写哦！或者用自己喜欢的单词填补右边的字典树！

核心特征1：多叉树结构

字典树与二叉树不同，它的每个节点可以有多个子节点。每个节点的子节点数量，由可能出现的字符数量决定，这些可能出现的字符集合，我们称为“字符集”。在常见的英文字符处理中，字符集大小通常为 26（英文字母共26个）。字典树的每条路径，都唯一对应一个字符串，通过从根节点出发，沿着边所代表的字符顺序拼接，就能得到完整的字符串。

核心特征2：节点不存储字符（路径构成字符串）

在字典树中，节点主要用于引导路径走向，并不存储字符。这种设计让字典树在存储大量具有相同前缀的字符串时，能够节省大量空间。比如，存储“program”“programmer”“programming”时，这些单词共享“program”前缀，在字典树中只需要存储一次该前缀路径，大大减少了存储空间。

字典树与二叉树的对比

字典树与二叉树虽都是树，但二者有很大的不同：

- 子节点数量不固定

二叉树每个节点最多有两个子节点，而字典树每个节点的子节点数量，取决于字符集的大小。以英文字母为例，字典树每个节点最多可能有26个子节点。这种差异使得字典树在处理字符串时更具灵活性，能更自然地表示字符之间的关系。

- 路径具有明确语义

二叉树的路径通常没有特定语义，主要用于搜索和排序操作。而字典树的每条路径都代表一个有意义的字符串，这使得字典树在字符串查找、前缀匹配等操作上效率极高。例如，要查找以“pre”为前缀的所有单词，只需在字典树中找到“pre”对应的路径，然后遍历该路径下的所有子树，就能找到所有符合条件的单词。

2.3.2 字典树的性质

前面我们已经对字典树的定义、核心特征，以及与二叉树的区别有了清晰的认识。接下来，我们将一起深入探究字典树在时间和空间层面展现出的独特性质，这将帮助大家更好地理解和运用这一数据结构。

时间复杂度特性

字典树在插入和查询操作上，有着出色的时间复杂度表现，均只需 $O(|S|)$ 。这里的 $|S|$ ，指的是要插入或查询的字符串的长度。

在插入操作中，当我们要向字典树中插入一个新字符串时，从根节点开始，按照字符串中字符的顺序，沿着对应的边依次向下遍历。如果某条边不存在，就创建一条新边。例如，向字典树中插入单词“banana”，从根节点出发，找到代表“b”的边，若该边存在则顺着它移动到下一个节点；若不存在，就创建一条新的“b”边。接着对“a”“n”等后续字符执行同样操作，直到整个单词插入完成。由于这个过程与字符串的长度成正比，因此插入操作的时间复杂度为 $O(|S|)$ 。

查询操作的过程与插入类似。同样从根节点出发，根据待查询字符串的字符顺序，沿着相应的边进行遍历。因为查询过程同样只需要遍历字符串的每个字符一次，所以查询操作的时间复杂度也是 $O(|S|)$ 。

空间特性

字典树在空间利用上非常高效，这主要得益于它能让不同字符串共享相同的前缀。如当字典树中同时存储“apple”和“app”这两个单词时，由于它们前三个字符“app”是相同的，在字典树里，这部分前缀只需要存储一次。这种共享前缀的特性，在存储大量具有相同前缀的字符串时，优势尤为明显。当在存储包含“program”前缀的众多单词，“program”这部分前缀只需存储一次，大大节省了存储空间。

然而，当字符串之间的公共前缀较少时，字典树的空间优势就会减弱，甚至可能会占用较多的空间。

2.3.3 字典树的实现

前面我们详细了解了字典树的定义、特性，现在就来动手实现字典树的基本功能，包括初始化、插入和查询操作。我们将使用C++语言来完成代码编写，通过具体的代码实现，加深对字典树的理解。

步骤1：定义存储结构

```
1 const int N = 1e5+10, M = 26; // N:最大结点数 M:字符集大小
2 int trie[N][M]; // trie[p][c]表示p结点通过字符c到达的结点编号
3 bool isEnd[N]; // 记录结点是否为单词结尾
4 int idx = 0; // 当前可分配的结点编号(从1开始)
5 return go(f, seed, [])
6 }
```

图2.3.3 定义存储结构

在这里，我们定义了一个二维数组 `trie` 来存储字典树的结构。`trie[p][c]` 表示从编号为 `p` 的节点，通过字符 `c` 所到达的下一个节点的编号。`isEnd` 是一个布尔类型的数组，用于标记某个节点是否是一个单词的结尾。`idx` 记录了当前可以分配的新节点的编号，初始时为 0，并且我们约定从 1 开始分配新节点。

步骤2：进行初始化

```
1 void init() {
2     memset(trie[0], 0, sizeof(trie[0])); // 根结点编号为0
3     idx = 1; // 下一个可用结点从1开始分配
4 }
```

图2.3.4 进行初始化

在 `init` 函数中，我们使用 `memset` 函数将 `trie` 数组中根节点（编号为 0）的所有子节点编号初始化为 0，表示根节点刚开始没有任何子节点。同时，将 `idx` 设置为 1，以便下一次创建新节点时从编号 1 开始。

插入操作实现

在 `insert` 函数中，我们从根节点（编号为 0）开始，遍历要插入的字符串 `s` 的每个字符。将字符转换为对应的数字（例如 'a' 转换为 0，'b' 转换为 1 等），检查当前节点 `p` 是否存在对应字符的子节点。如果不存在，就初始化一个新

节点，并将当前节点 p 的对应子节点编号设置为新节点的编号，同时更新 idx 。然后移动到新的子节点继续处理下一个字符。当整个字符串处理完毕后，将最后一个节点标记为单词结尾。如图2.3.5的流程图所示，更加清晰的看到插入是如何操作的。

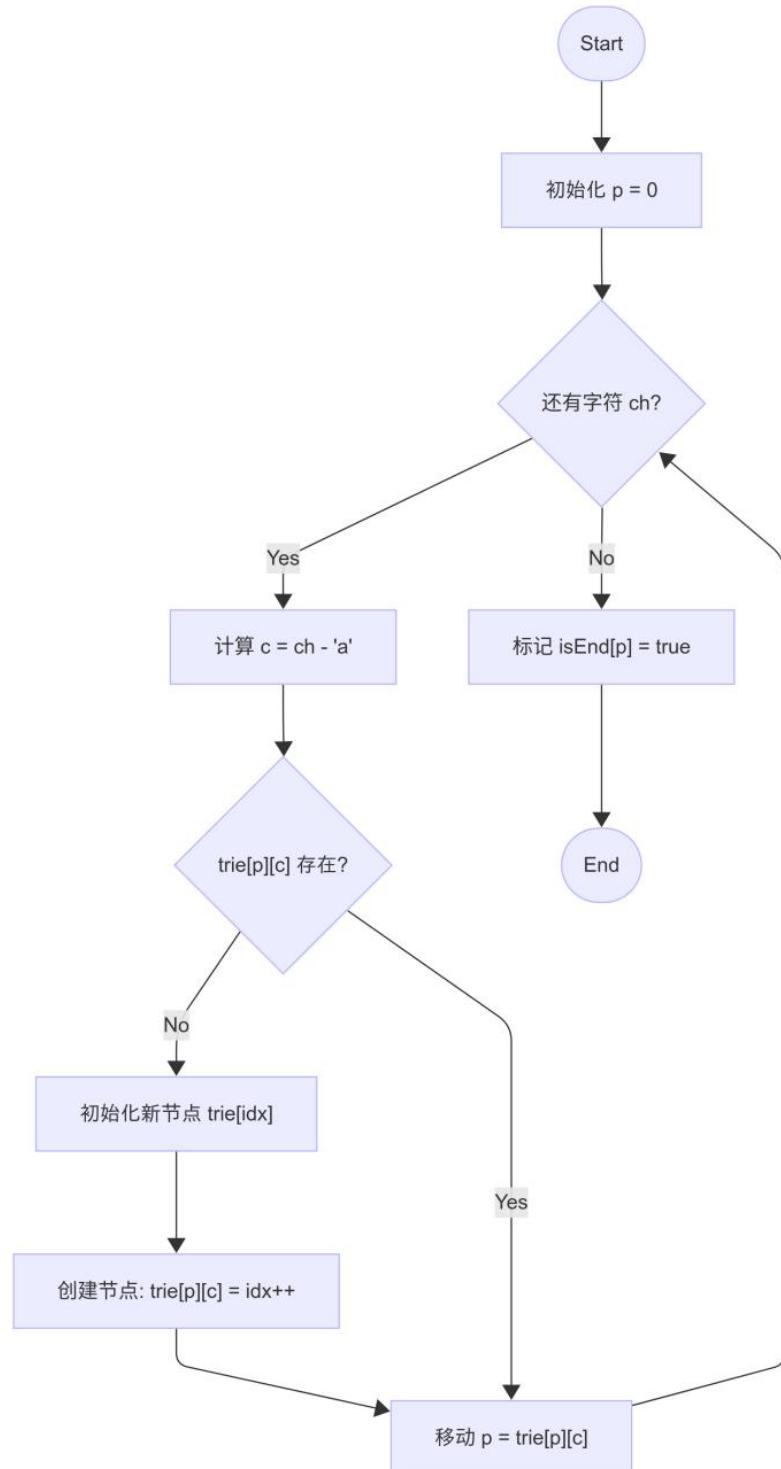


图2.3.5 插入操作流程图

插入操作是将一个字符串插入到字典树中，具体代码如下：

```
1 void insert(string s) {
2     int p = 0; // 从根结点出发
3     for(char ch : s) {
4         int c = ch - 'a'; // 字符转数字
5         if(!trie[p][c]) { // 没有对应子结点
6             memset(trie[idx], 0, sizeof(trie[idx])); // 初始化新结点
7             trie[p][c] = idx++; // 创建新结点
8         }
9         p = trie[p][c]; // 移动到子结点
10    }
11    isEnd[p] = true; // 标记单词结尾
12 }
```

图2.3.6 插入操作代码

查询操作实现

在 `search` 函数中，同样从根节点开始，遍历要查询的字符串 `s` 的每个字符。将字符转换为数字后，检查当前节点 `p` 是否存在对应字符的子节点。如果不存在，直接返回 `false`，表示该字符串不存在于字典树中。如果能顺利遍历完整个字符串，最后检查最后一个节点是否被标记为单词结尾，以此来确定该字符串是否精确存在于字典树中。

查询操作是判断一个字符串是否存在字典树中，代码如下：

```
1 bool search(string s) {
2     int p = 0;
3     for(char ch : s) {
4         int c = ch - 'a';
5         if(!trie[p][c]) return false;
6         p = trie[p][c];
7     }
8     return isEnd[p]; // 必须精确匹配
9 }
```

图2.3.7 查询操作代码

通过以上代码的实现，我们初步完成了字典树的基本功能。大家可以在实际编程中运用这些代码，处理字符串相关的问题，进一步体会字典树的优势和特点。

2.3.4 字典树的应用

接下来我们来试着用所学到的知识解出下面的问题。

题目描述

某校长任命你为特派探员，每天记录他的点名。校长会提供化学竞赛学生的人数和名单，而你需要告诉校长他有没有点错名。

输入格式：

第一行一个整数 n ，表示班上人数。接下来 n 行，每行一个字符串表示其名字（互不相同，且只含小写字母，长度不超过50）。第 $n+2$ 行一个整数 m ，表示教练报的名字个数。接下来 m 行，每行一个字符串表示教练报的名字（只含小写字母，且长度不超过 50）。

输出格式：

对于每个教练报的名字，输出一行。如果该名字正确且是第一次出现，输出 `OK`，如果该名字错误，输出 `WRONG`，如果该名字正确但不是第一次出现，输出 `REPEAT`。

步骤1：构建字典树

首先定义字典树的节点类，每个节点包含一个字典用于存储子节点（键为字符，值为子节点对象），以及一个布尔值表示该节点是否是一个单词的结尾。

遍历班上学生的名单，将每个学生的名字插入到字典树中。插入过程是从根节点开始，依次处理名字中的每个字符，如果当前字符对应的子节点不存在，则创建一个新的子节点，然后移动到该子节点继续处理下一个字符，直到名字的所有字符处理完，并将最后一个节点标记为单词的结尾。

步骤2：处理校长点名

对于校长报出的每个名字，从字典树的根节点开始进行查找。依次处理名字中的每个字符，检查当前字符对应的子节点是否存在。如果不存在，说明这个名字不在字典树中，即名字错误，输出 WRONG。如果当前字符对应的子节点存在，继续处理下一个字符，直到名字的所有字符处理完。此时检查最后一个节点

是否标记为单词的结尾，如果不是单词的结尾，也说明名字错误，输出 WRONG；如果是单词的结尾，说明名字是正确的。

为了判断名字是否是第一次出现，需要额外使用一个数据结构（如集合）来记录已经出现过的名字。当确定名字正确后，检查该名字是否在记录已出现名字的集合中。如果不在，输出 OK，并将该名字加入到集合中；如果在，说明该名字不是第一次出现，输出 REPEAT。

步骤3：重复上述步骤

对校长报出的所有名字，都重复上述查找和判断的过程，直到所有名字处理完毕。

通过以上步骤，利用字典树的特性来高效地存储和查找学生名字，从而实现对校长点名情况的准确判断。

核心代码展示

```
1 int cnt[N]; // 新增计数数组
2
3 void insert(string s) {
4     int p = 0;
5     for(char ch : s) {
6         int c = ch - 'a';
7         if(!trie[p][c]) trie[p][c] = idx++;
8         p = trie[p][c];
9         cnt[p]++;
10    }
11    isEnd[p] = true;
12 }
```

图2.3.8 统计前缀出现次数

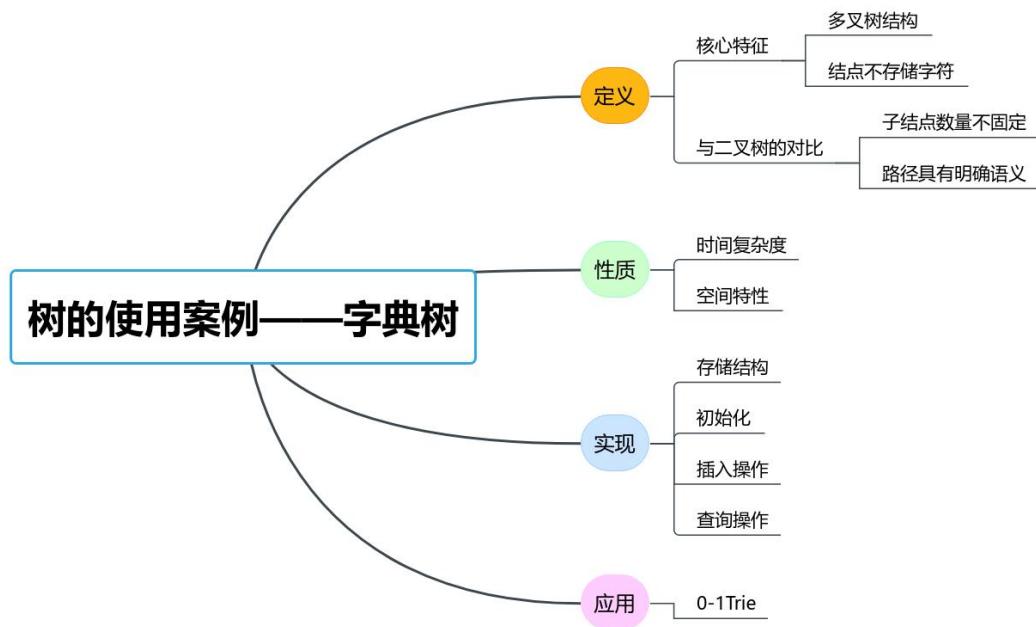


拓展提升：0-1Trie

01-trie 是指字符集为 {0, 1} 的 trie。将数的二进制表示看做一个字符串，就可以建出01-trie。trie可以方便的统计在某一位上某种数字出现了多少次。因此，这个数据结构可以用于异或极值、异或和等与出现次数有关的问题。



知识图谱

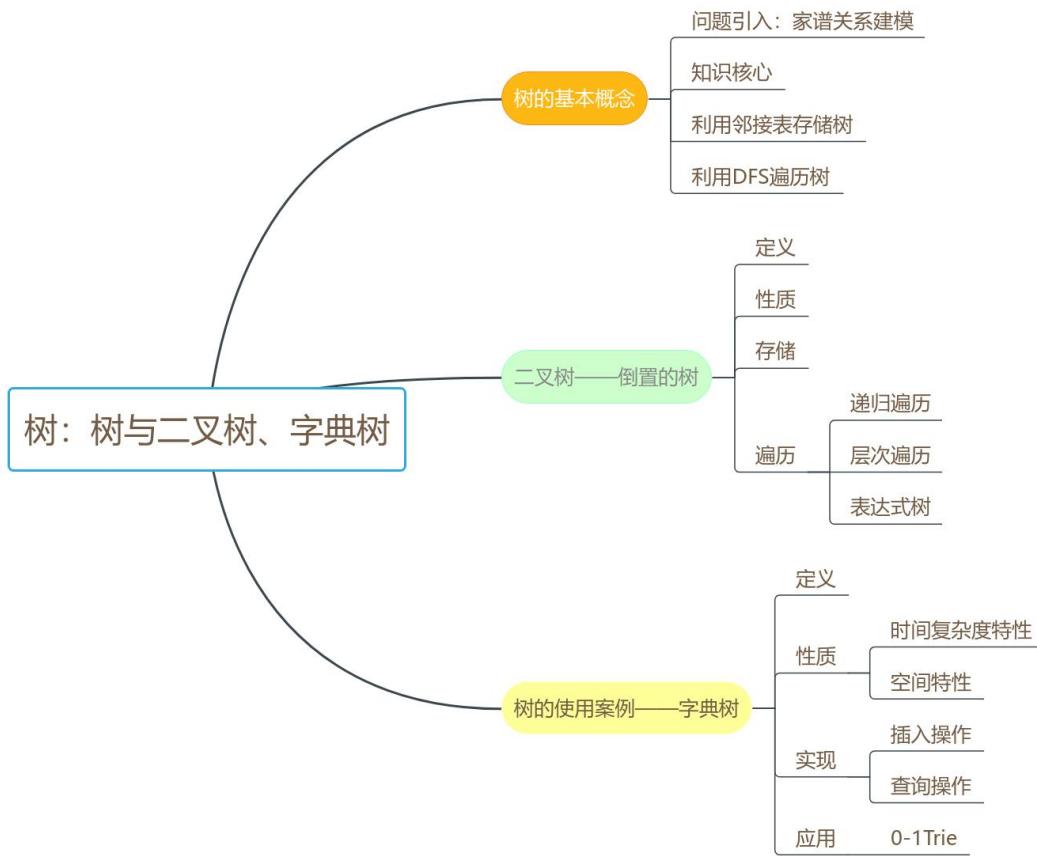


练习提升

1. 给定一棵 n 个点的带权树，结点下标从 1 开始到 n 。寻找树中找两个结点，求最长的异或路径。（注：异或路径指的是指两个结点之间唯一路径上的所有边权的异或。）
2. 给定一个字典树和一系列单词，编写一个程序来将这些单词插入到字典树中。
假设字典树中的每个节点存储一个字符，并且每个节点有一个布尔值 `isEnd` 来标记单词的结尾。
 - 具体要求：
 - 2.1 编写一个 `insert` 函数，该函数接收一个字符串 s 作为参数。
 - 2.2 从字典树的根节点开始，逐个字符地检查字符串 s 中的字符。
 - 2.3 如果字符对应的子节点不存在，则创建一个新的子节点，并将其链接到当前节点。
 - 2.4 更新 `isEnd` 标记，以指示字符串 s 的结尾。

总结 评价

1. 下图展示了本章的核心概念与关键能力，请同学们对照图中的内容进行总结。



2. 根据自己的掌握情况填写下表。

学习内容	掌握程度
树的基本概念	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
用代码实现树的遍历	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
二叉树的定义和关键术语	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
二叉树的遍历	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
字典树的基本特征与实现	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
字典树的应用	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解

小结

在本章中，我们围绕树这一核心主题展开了深入探索。通过剖析树形结构的层次关系与递归特性，我们不仅掌握了二叉树的前序、中序、后序遍历等基础操作，更通过竞赛真题理解了完全二叉树、满二叉树等特殊结构的应用场景。在字典树的专题中，我们通过字符串匹配、前缀统计等经典问题，领略了这种高效数据结构在信息检索中的独特优势。本章内容着重培养从问题特征出发选择数据结构的能力——面对需要体现层次关系、快速前缀匹配或递归求解的问题时，树结构往往能提供更优雅的解决方案。希望同学们能将二叉树的递归思维与字典树的空间优化技巧融会贯通，在面对搜索路径优化、字符串处理、最优决策树构建等复杂问题时，展现出更扎实的算法设计与问题拆解能力，为后续高阶数据结构的学习奠定坚实基础。

第三章

树的应用：二叉搜索树与并查集

前面我们已学习了基础数据结构，它们在解决简单问题时游刃有余。但当遇到更复杂的竞赛题目，如涉及数据高效检索、集合动态合并与查询等问题时，就需借助更强大的数据结构。

树，作为一种重要的数据结构，以其独特的层次关系，在众多领域发挥着关键作用。本章我们将深入探讨树的几种重要应用：二叉搜索树能高效地进行数据查找、插入和删除；堆可轻松实现优先队列；并查集在处理集合的合并与查询时十分高效；最近公共祖先（LCA）则能帮助我们快速解决树上节点关系的问题。

二叉搜索树（BST）如同一位严谨的图书管理员，通过其独特的排序特性，将数据整理得井然有序。这种与生俱来的秩序感，使得它在数据库索引、字典实现等场景中游刃有余，让 $O(\log n)$ 时间复杂度的搜索梦想照进现实。

而并查集（Disjoint Set Union）则像一位精于社交网络分析的关系大师，用树结构编织出高效的连通性判断网络。其路径压缩与按秩合并的智慧，让社交网络中的好友关系判断等问题，在近乎常数时间内得到优雅解答。

本章将带领读者深入这两个经典数据结构的核心：从BST如何通过旋转保持平衡蜕变为AVL树，到并查集如何在Kruskal算法中搭建最小生成树；从理论推导到代码实现，我们将在平衡与效率的博弈中，见证树结构如何将抽象数学之美转化为解决实际工程难题的利刃。准备好开启这场融合算法智慧与现实应用的探索之旅了吗？



3.1 二叉树的应用案例——堆

本节学习目标

- ◆ 理解堆的定义与核心性质，区分大根堆与小根堆的序性规则，明确二叉堆作为完全二叉树的存储特性。
- ◆ 掌握堆的操作流程与复杂度分析，手动模拟插入（自底向上交换）和删除（根节点下沉）的完整调整逻辑。
- ◆ 应用堆模型解决实际问题，编写堆的数组实现代码，理解索引映射（父节点、左右子节点计算）及调整终止条件。

3.1.1 堆的定义与基本性质

堆的定义

堆（**Heap**）是一类具有特殊序性的树状数据结构，其核心特征是父节点的值始终大于或小于其子节点的值。根据根节点存储的元素类型，堆可分为两种：

- 大根堆：根节点为全堆最大值，且每个父节点均大于或等于其子节点。
- 小根堆：根节点为全堆最小值，且每个父节点均小于或等于其子节点。

作为堆的高效实现形式，二叉堆（**Binary Heap**）¹是一棵满足堆序性的完全二叉树。

堆的基本性质

基于堆的序性规则，可进一步推导出以下重要性质：

- 子树的自堆性

堆的递归定义决定了其子结构的序性。若根节点满足堆序性，则其左右子树各自独立构成子堆。例如，在最大堆中，根节点的左子树所有节点均小于根节点，

¹ 若无特殊说明，本书中“堆”均指二叉堆。

且左子树内部仍满足父节点 \geq 子节点的性质；同理适用于右子树。这一性质为堆的局部调整操作（如插入、删除后的修复）提供了理论依据。

• 结构多样性的成因

堆的完全二叉树形态允许不同排列方式，只要满足堆序性。以数据集合[7, 6, 5, 4, 3, 2, 1] 构建最大堆为例，存在两种合法结构（互为镜像）：

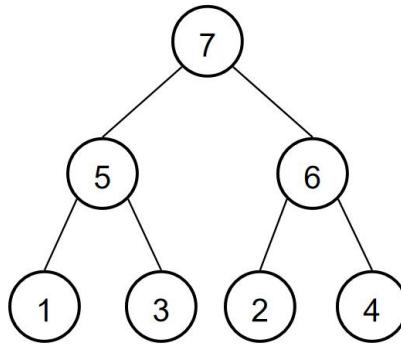


图3.1.1 原堆

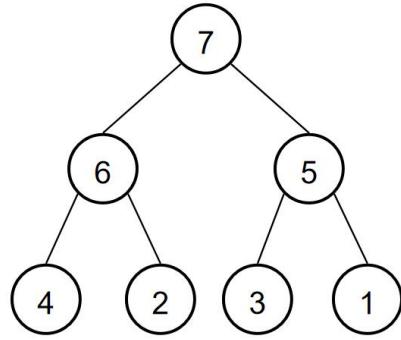


图3.1.2 镜像堆

3.1.2 堆的核心操作与复杂度分析

堆的高效性依赖于两个关键操作：插入元素和删除元素。其调整过程均通过交换父子节点实现，但方向相反，分别称为上浮（插入）和下沉（删除）。

插入操作

1. 放置新元素

将新元素添加到完全二叉树的最底层最右侧，作为新的叶子节点。（例如，若堆当前有k层，新元素将位于第k+1层的最右位置。）

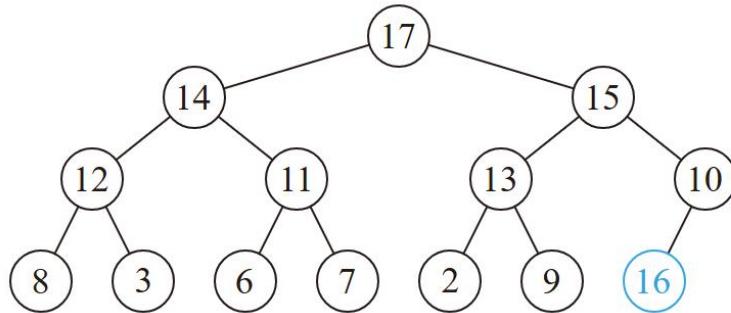


图3.1.3 放置新元素

2. 自底向上调整（上浮）

若新节点的值大于其父节点（大根堆）或小于其父节点（小根堆），则交换两者位置。重复上述比较，沿父节点路径向上调整，直至到达根节点或父节点满足堆序性。每次交换后，新节点所在的子树必然满足堆性质。因为交换前父节点已符合堆序性，交换后新节点作为子树的根，其子节点（原父节点的兄弟）必然小于/大于新节点。

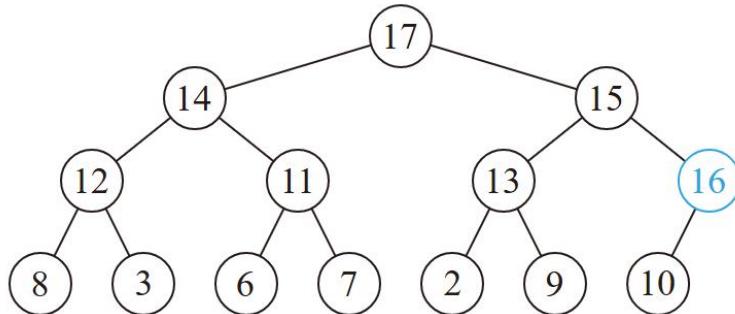


图3.1.4 自底向上调整

3. 终止条件

当新节点到达根节点，或无需交换时，插入操作完成。

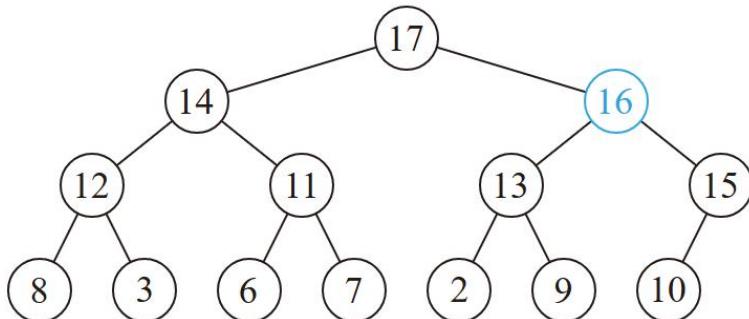


图3.1.5 插入操作完成

删除操作

1. 交换根与末元素

将根节点（最大值/最小值）与最底层最右侧的叶子节点交换，确保堆结构仍为完全二叉树。

2. 删除末节点

移除原末节点（即待删除元素），此时堆的节点数减1。

3. 自顶向下调整（下沉）

比较根节点的左右子节点（若存在），选择较大者（大根堆）或较小者（小根堆）。若子节点值大于/小于根节点，则交换两者位置，并继续对交换后的子树执行相同操作。未参与交换的子树（如右子树未参与时，左子树保持原结构）必然符合堆性质，因为调整前根节点已满足堆序性，且交换仅影响局部子树。

4. 终止条件

当根节点到达叶子层，或无需交换时，删除操作完成。

时间复杂度分析

最坏情况下，新元素需从底层叶子交换至根节点，路径长度为树的高度 $O(\log n)$ 。每层仅需常数次比较和交换，故时间复杂度为 $O(\log n)$ 。



思考辨析

删除操作的时间复杂度是多少？为什么？

删除根节点后，可能需要从根节点交换至叶子层，路径长度同样为 $O(\log n)$ 。可以将删除看作插入的逆向过程，尽管调整方向与插入相反，但路径长度相同，因此时间复杂度亦为 $O(\log n)$ 。

例题练习

给定一个数列，初始为空，请支持下面三种操作：

1. 给定一个整数 x , 请将 x 加入到数列中。
2. 输出数列中最小的数。
3. 删除数列中最小的数 (如果有多个数最小, 只删除 1 个)。

关键代码如下:

```
1 const int N = 1e6 + 2;
2 struct Heap
3 {
4     int a[N];
5     int size = 0;
6 #define father(x) (x >> 1)
7 #define ls(x) (x << 1)
8 #define rs(x) (ls(x) + 1)
9     void push(int x)
10    {
11        int pos = ++size;
12        a[pos] = x;
13        while (father(pos) != 0)
14        {
15            if (a[pos] < a[father(pos)])
16            {
17                swap(a[pos], a[father(pos)]);
18                pos = father(pos);
19            }
20            else
21                break;
22        }
23    }
24    void pop()
25    {
26        swap(a[1], a[size--]);
27        int pos = 1;
28        while (ls(pos) <= size)
29        {
30            if (rs(pos) <= size && a[rs(pos)] < a[ls(pos)])
31                pos = rs(pos);
32            else
33                pos = ls(pos);
34            if (a[father(pos)] < a[pos])
35                return;
36            swap(a[father(pos)], a[pos]);
37        }
38    }
39    int top()
40    {
41        return a[1];
42    }
43};
```

图3.1.6 数列操作题关键代码

3.1.3 STL中的堆实现

C++标准模板库（STL）提供了 `priority_queue` 容器，用于高效实现堆结构。其底层默认采用最大堆序性，但可通过模板参数灵活配置为小根堆。

```
1 priority_queue<int> q1; // 默认是大根堆  
2 priority_queue<int, std::deque<int>, std::greater<int>> q2; // 小根堆
```

图3.1.7 STL中的堆实现

成员函数与复杂度分析

`priority_queue` 提供以下核心操作，按时间复杂度分类：

1. 常数时间复杂度操作 ($O(1)$)

`top()`: 访问堆顶元素（最大值/最小值），要求队列非空。

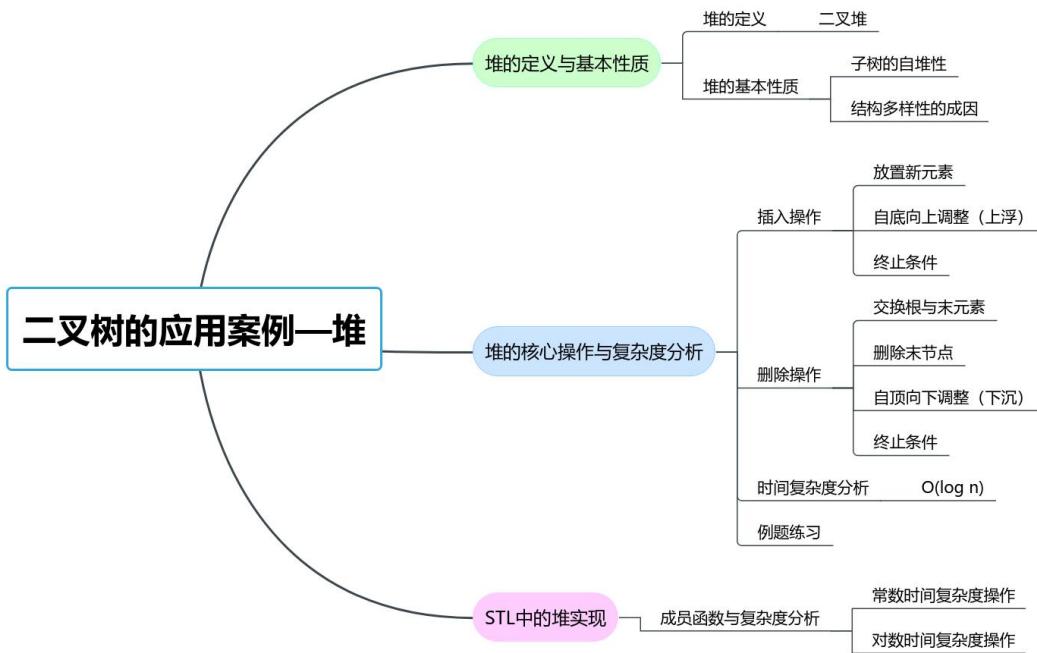
`empty()`: 检查队列是否为空。

`size()`: 返回队列中元素数量。

2. 对数时间复杂度操作 ($O(\log n)$)

`push(x)`: 插入新元素x，自动调整堆结构以维持序性。

`pop()`: 删除堆顶元素，将底层容器末元素移至堆顶并下沉调整。



一、判断题

1. 二叉堆的根节点一定是全堆的最小值。
2. 在最大堆中插入新元素后，只需调整根节点即可恢复堆性质。

二、选择题

1. 下列选项中，符合堆性质的结构是？
 - A. 根节点为5，左子节点为3，右子节点为7
 - B. 根节点为5，左子节点为7，右子节点为3
 - C. 根节点为5，左子节点为5，右子节点为5
 - D. 以上均符合
2. 删除堆顶元素后，堆调整的方式是？
 - A. 将末元素移至堆顶，然后向下交换
 - B. 将堆顶元素与末元素交换后删除
 - C. 直接删除堆顶元素，无需调整

D. 重新构建整个堆

三、填空题

1. 在空堆中依次插入元素3、1、5，形成最大堆后，根节点的值为_____。
2. 堆的时间复杂度主要取决于树的____，其值为_____。

四、简答题

1. 简述堆的"子树自堆性"对插入操作的意义。
2. 为什么删除堆顶元素后，未参与交换的子树仍保持堆性质？

3.2 森林的应用案例——并查集

本节学习目标

- ◆ 理解并查集的核心原理与结构，掌握森林结构表示集合的思想，能解释合并与查询的作用。
- ◆ 实现并查集的基础操作与优化策略，能独立编写初始化、路径压缩和启发式合并的代码。
- ◆ 应用并查集解决实际问题并分析复杂度，能将连通性问题抽象为集合操作，设计解决方案。

3.2.1 并查集的定义及核心操作

并查集的定义

在上一节课中，我们已经学习了关于森林的知识。那么，如何将这些知识应用到实际中呢？让我们欢迎本节课的主角——并查集登场。并查集是一种高效的数据结构，用于管理元素的集合归属问题。它以森林的形式实现，每棵树代表一个集合，树中的每个节点则代表集合中的一个元素。

并查集的核心操作

并查集的核心操作包括两个方面：查询元素所属的集合，以及合并两个集合。

查询（Find）：查询某个元素所属集合（查询对应的树的根节点），这可以用于判断两个元素是否属于同一集合。

合并（Union）：合并两个元素所属集合（合并对应的树）

我们将属于同一集合的元素置于同一棵树内。因此，若两个元素拥有相同的根节点，则它们属于同一个集合；反之，若它们的根节点不同，则它们不属于同一个集合。如图3.2.1。

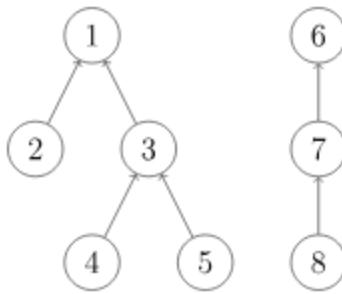


图3.2.1 查询与合并

3.2.2 核心操作解析

初始化

在一开始，所有元素都属于一个单独的集合，且不属于其他的集合。在实现上，我们让所有节点都形成一个只有一个节点的树。

```

void init(int n) {
    for (int i = 1; i <= n; ++i) {
        parent[i] = i;
    }
}

```

图3.2.2 初始化代码

查询操作

要想查询一个元素的根节点，只需要不断向他的父节点“跳”就可以了。根据上面初始化的实现，如果某一节点的父节点是他自己，那么这个节点是一个根节点。那如果两个节点具有相同的根节点，那么他们就属于同一集合。

```

int find(int x) {
    if (parent[x] == x) return x;
    return find(parent[x]);
}

// 判断是否属于同一集合
bool same(int x, int y) {
    return find(x) == find(y);
}

```

图3.2.3 查询操作代码

优化查询——路径压缩

在一次查询过程中，经过的每个元素都属于当前查询的集合，我们可以将其直接连到根节点以加快后续查询。

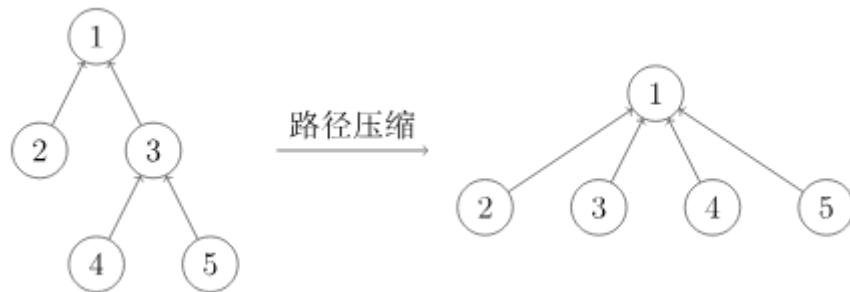


图3.2.4 路径压缩示意图

具体代码如图3.2.6，大家可以仔细体会优化前和优化后的区别和优劣！

```
int find(int x) {
    if (parent[x] == x) return x;
    return parent[x] = find(parent[x]);
}
```

图3.2.5 查询代码优化

合并操作

合并时，选择哪棵树的根节点作为新树的根节点会影响未来操作的复杂度。因此我们可以将节点较少或深度较小的树连到另一棵，以免发生退化。

```
void merge(int x, int y) {
    x = find(x);
    y = find(y);
    if (x == y) return;
    parent[y] = x;
}
```

图3.2.6 合并操作代码

优化合并——启发式合并

要想合并两个集合，也只要让他们所属的树具有相同的根即可。因此在查询出属于同一个集合的节点后，接下来就是进行合并。

```
int size[N];
void merge(size_t x, size_t y) {
    x = find(x);
    y = find(y);
    if (x == y) return;
    if (size[x] < size[y]) swap(x, y);
    parent[y] = x;
    size[x] += size[y];
}
```

图3.2.7 合并代码优化

具体代码如图3.2.7，大家可以再仔细体会优化前和优化后的区别和优劣！并思考为什么查询和合并可以这样优化。

3.2.3 时间复杂度分析

根据我们在前几节学到的知识，你也许会觉得未经优化的查询的时间复杂度是 $O(\log n)$ ，这是因为一次操作发生的“跳”的次数不会超过树的高度。然而，这只是我们理想中的复杂度。

事实上，如果使用未经优化的合并，你最后有可能会得到若干条很长的链，这时复杂度就会超过 $O(\log n)$ 。同时使用路径压缩和启发式合并之后，并查集的每个操作平均时间仅为 $O(\alpha(n))$ ，其中 $\alpha(n)$ 为阿克曼函数的反函数，其增长极其缓慢，也就是说其单次操作的平均运行时间可以认为是一个很小的常数。

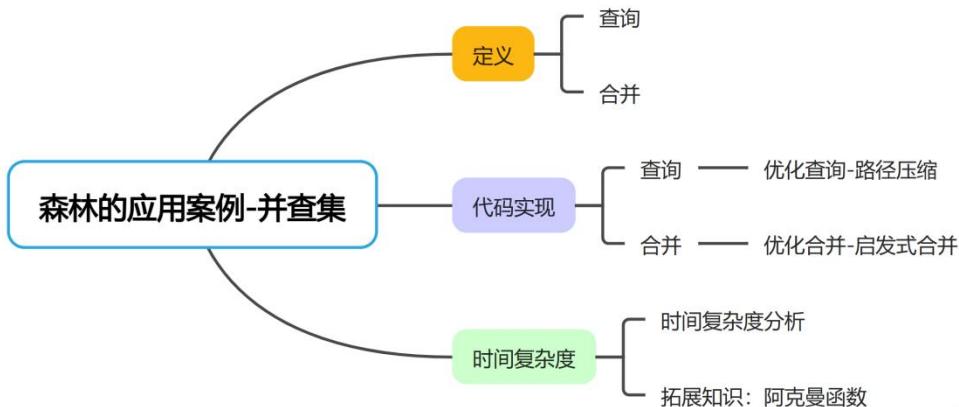


拓展知识：阿克曼函数

阿克曼函数 $A(m,n)$ 的定义是这样的：

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{otherwise} \end{cases}$$

而反阿克曼函数 $\alpha(m,n)$ 的定义是阿克曼函数的反函数，即为最大的整数 m 使得 $A(m,m) \leq n$ 。



一、判断题

1. 并查集是一种用于处理动态连通性问题的数据结构，它只能用于无向图。
2. 在并查集中，路径压缩操作可以显著提高查找操作的效率。
3. 并查集只能用于判断两个元素是否在同一集合中，不能用于计算集合的大小。
4. 并查集的合并操作（Union）必须按照元素的大小顺序进行。
5. 并查集的时间复杂度为 $O(1)$ 。

二、选择题

1. 在并查集中，路径压缩操作的作用是什么？
 - A. 增加树的高度
 - B. 减少树的高度
 - C. 增加集合的数量
 - D. 减少集合的数量
2. 并查集的按秩合并（Union by Rank）操作的主要目的是什么？
 - A. 保持树的平衡
 - B. 增加树的高度
 - C. 减少树的节点数量
 - D. 增加集合的大小

3. 在并查集中，假设集合中有 n 个元素，经过若干次合并操作后，集合的总数最多为：

- A. n
- B. $n/2$
- C. 1
- D. 无法确定

三、应用题

在一个社交网络中，有 n 个人，编号从 1 到 n 。现在有若干条关系，每条关系表示两个人是朋友。如果 A 和 B 是朋友，B 和 C 是朋友，那么 A 和 C 也被认为是朋友。请判断任意两个人是否属于同一个社交圈。

输入：

第一行包含两个整数 n 和 m ，分别表示人数和关系数。

接下来 m 行，每行包含两个整数 u 和 v ，表示 u 和 v 是朋友。

最后一行包含两个整数 x 和 y ，表示查询 x 和 y 是否属于同一个社交圈。

输出：

如果 x 和 y 属于同一个社交圈，输出“YES”；否则输出“NO”。

3.3 树的应用案例——最近公共祖先（LCA）

本节学习目标

- ◆ 掌握LCA与倍增法核心原理，理解最近公共祖先的定义及性质，明确其在树结构中的关键作用。
- ◆ 实现预处理与查询完整流程，独立完成LCA查询的两阶段步骤（深度对齐、共同跳跃），分析预处理与查询的时间复杂度。
- ◆ 应用LCA解决实际问题，编写并调试LCA查询代码，处理输入输出格式，验证代码正确性。

3.3.1 问题引入：从并查集到树结构的延伸性质

在 3.2 节中，我们学习了“并查集”——一种通过森林结构管理元素集合的高效数据结构，其核心是通过路径压缩和启发式合并优化查询与操作效率。本节我们将视角转向树结构的另一个经典问题：**最近公共祖先（LCA）**。尽管并查集与 LCA 的应用场景不同（前者处理集合关系，后者解决树上的祖先查询），但两者的设计思想有共通之处：通过预处理与优化操作，将线性复杂度降至对数级别。

首先让我们来看一个实际问题：给定一棵包含n个节点的树，进行m次查询，每次查询两个节点u和v的最近公共祖先（LCA）。

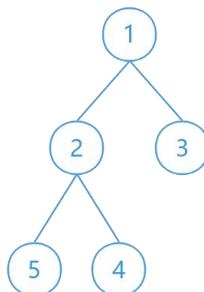


图3.3.1 树示意图

在图示树结构中， $LCA(4,5) = 2$ ， $LCA(4,3) = 1$ 。

3.3.2 核心概念与倍增法原理

LCA的定义与性质

接下来，让我们了解一下LCA的定义以及性质，这部分内容会帮助我们进行后续的问题解决。

1. 严格定义

给定一棵有根树和两个结点 u 和 v ，其最近公共祖先（LCA）是满足以下条件的结点 w ：

- ① w 是 u 和 v 的公共祖先（即 w 在从根到 u 和从根到 v 的路径上）。
- ② 在所有公共祖先中， w 的深度最大（离根最远）。

2. 关键性质

主要包括以下三个性质：

性质	描述	应用场景
祖先优先性	若 u 是 v 的祖先，则 $LCA(u,v)=u$	快速判定父子关系
路径必经性	u 到 v 的最短路径必经过其 LCA	计算树上两点距离
结合律	$LCA(u,LCA(v,w))=LCA(LCA(u,v),w)$	处理多结点 LCA

表3-1 LCA的性质

朴素算法的问题与倍增法优化

1. 暴力回溯法的瓶颈

问题场景：

假设要在一棵包含 10^5 个节点的树上进行 10^5 次查询，暴力回溯法的总操作次数将达到 10^{10} ，远超程序的时间承受能力（通常竞赛程序需在1秒内完成约 10^8 次操作）。

而暴力回溯法效率低下的原因如下：

- ① 线性回溯：每次查询需从结点逐层回溯到根，路径长度与树高成正比。
- ② 重复计算：不同查询的路径可能存在重叠，但暴力法未利用这一特性。

2. 倍增法的优化思想

核心策略：

空间换时间——通过预处理存储每个结点的多级祖先信息，将查询时的线性跳跃优化为对数级跳跃。

具体设计如下：

- ① 二进制拆分：

将跳跃步长分解为 2^k 的指数形式（如1步、2步、4步...）。好比于电梯设置 2^k 层的快速按钮（如1层、2层、4层），通过组合按钮快速到达任意楼层。

- ② 跳跃表预处理：

为每个结点x预存其 2^i 级祖先fa[x][i]，形成一张“跳跃地图”。

递推构建fa[x][i] = fa[fa[x][i-1]][i-1]，类似“跳2步 = 跳1步后再跳1步”。

3. 查询优化过程

• 步骤一：深度对齐

为了将较深结点u快速抬升至与v同深度，我们需要从最大可能的 2^k 步开始尝试跳跃，逐步逼近目标深度。

```
1 // 示例：若 u 比 v 深 5 层（二进制 101），则依次跳 4 层和 1 层
2 for (int k = 20; k >= 0; k--)
3     if (depth[u] - (1 << k) >= depth[v])
4         u = fa[u][k];
5 return go(f, seed, [])
6 }
```

图3.3.2 深度对齐

• 步骤二：同步跳跃找LCA

经过观察我们会发现若 u 和 v 的 2^k 级祖先不同，说明 LCA 在更高层，需继续跳跃，这个时候就需要从高位到低位同步调整 u 和 v ，直至找到公共祖先。

```
1 for (int k = 20; k >= 0; k--)  
2     if (fa[u][k] != fa[v][k])  
3         u = fa[u][k], v = fa[v][k];  
4 return fa[u][0]; // 最终父结点即为 LCA
```

图3.3.3 同步跳跃找LCA



知识链接：其他 LCA 解法速览

方法	核心思想	适用场景	时间复杂度
Tarjan法	离线处理 并查集	批量查询	$O(n+m\alpha(n))$
树链剖分	重链分解 路径跳跃	动态树操作 (如修改边权)	$O(\log n)$
倍增法	二进制拆分 预处理跳跃表	静态树高频在线查 询	$O(n \log n)$

表3-2 其他LCA解法速览图

3.3.3 倍增法实现步骤详解

预处理阶段

1. DFS遍历建树

目标：记录每个结点的 深度 和 直接父结点（即 2^{20} 级祖先）。

操作步骤：

① 初始化：根结点的深度为0（或1，依定义约定），父结点设为无效值
(图3.3.4 预处理代码)。

② 递归遍历：对每个结点，访问其子结点并记录信息。

2. 构建倍增表

目标：预计算每个结点 u 的 2^k 级祖先 ($k \geq 1$)。

递推公式：

$$fa[u][k] = fa[fa[u][k-1]][k-1]$$

操作步骤：

- ① 外层循环：按 k 从小到大递推 ($1 \leq k < \text{LOG}$)。
- ② 内层循环：遍历所有结点 u ，计算其 2^k 级祖先。

```
1 // 预处理部分 (DFS)
2 void dfs(int u, int father) {
3     fa[u][0] = father;
4     depth[u] = depth[father] + 1;
5     for (int k = 1; k <= log_max; k++) {
6         fa[u][k] = fa[fa[u][k-1]][k-1];
7     }
8     for (auto v : tree[u]) {
9         if (v != father) dfs(v, u);
10    }
11 }
```

图3.3.4 预处理代码

查询阶段

1. 深度对齐

目标：将较深的结点上移至与另一结点同深度。

操作步骤：

- ① 比较深度：确保 u 是较深的结点。
- ② 二进制拆分：从高位到低位枚举 k ，若 u 能向上跳 2^k 步而不超过 v 的深度，则跳跃。

2. 共同跳跃找 LCA

目标：从高位到低位尝试跳跃，寻找最近的公共祖先。

操作步骤：

- ① 同步跳跃：从最大步长 2^k 开始尝试，若 u 和 v 的祖先不同则跳跃。
- ② 最终定位：循环结束后， u 和 v 的父结点即为 LCA。

```
13 // 查询 LCA
14 int lca(int u, int v) {
15     if (depth[u] < depth[v]) swap(u, v);
16     for (int k = log_max; k >= 0; k--) {
17         if (depth[fa[u][k]] >= depth[v]) u = fa[u][k];
18     }
19     if (u == v) return u;
20     for (int k = log_max; k >= 0; k--) {
21         if (fa[u][k] != fa[v][k]) {
22             u = fa[u][k];
23             v = fa[v][k];
24         }
25     }
26     return fa[u][0];
27 }
```

图3.3.5 查询LCA代码



思考辨析

为什么要假定两节点不同？

因为如果两个节点相同，LCA显然就是两个节点中深度小的那个。在最近公共祖先（LCA）问题中，假定两节点不同是算法实现的约定性前提：当两节点重合时，LCA即该节点本身（符合数学定义）；通过显式处理此情况可优化算法效率（避免无效遍历）、确保终止条件正确性，并规避潜在输入错误或特殊语义问题。

3.3.4 倍增法实现步骤详解

关键点总结

1. 倍增法的核心优势

维度	描述
时间复杂度	预处理 $O(n \log n)$, 单次查询 $O(\log n)$, 适合高频查询场景
空间复杂度	$O(n \log n)$, 需预存跳跃表
适用场景	静态树、在线查询（无需提前知道所有查询）

表3-3 倍增法的核心优势

2. 核心技巧

- ① 二进制拆分：将线性跳跃分解为 2^k 步长，利用预处理信息快速跳转。
- ② 深度对齐：通过二进制位运算对齐两结点深度，减少无效跳跃次数。
- ③ 同步跳跃：从高位到低位枚举步长，避免线性遍历祖先链。



扩展应用：计算树上两点距离

公式推导

树上两点 u 和 v 的路径长度可通过 LCA 快速计算：

$$d(u,v) = \text{depth}[u] + \text{depth}[v] - 2 \times \text{depth}[\text{LCA}(u,v)]$$

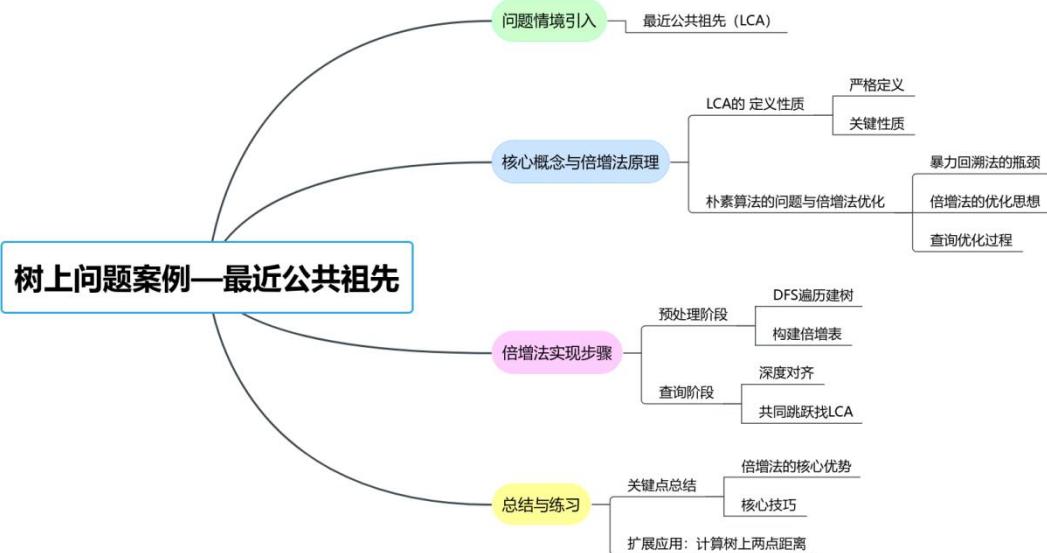
u 到 LCA 的路径长度为 $\text{depth}[u] - \text{depth}[\text{LCA}]$ 。

v 到 LCA 的路径长度为 $\text{depth}[v] - \text{depth}[\text{LCA}]$ 。

总路径长度为两者之和。



知识图谱



练习提升

一、基础题

1. 给定一棵完全二叉树，根为 1 号结点，求以下结点对的 LCA 和距离：

(7, 8)

(5, 9)

(3, 6)

2. 证明：在完全二叉树中，结点 i 的父结点为 $\lfloor i/2 \rfloor$ 。

二、进阶题

1. 实现倍增法求解 LCA，并通过洛谷 P3379 提交测试。

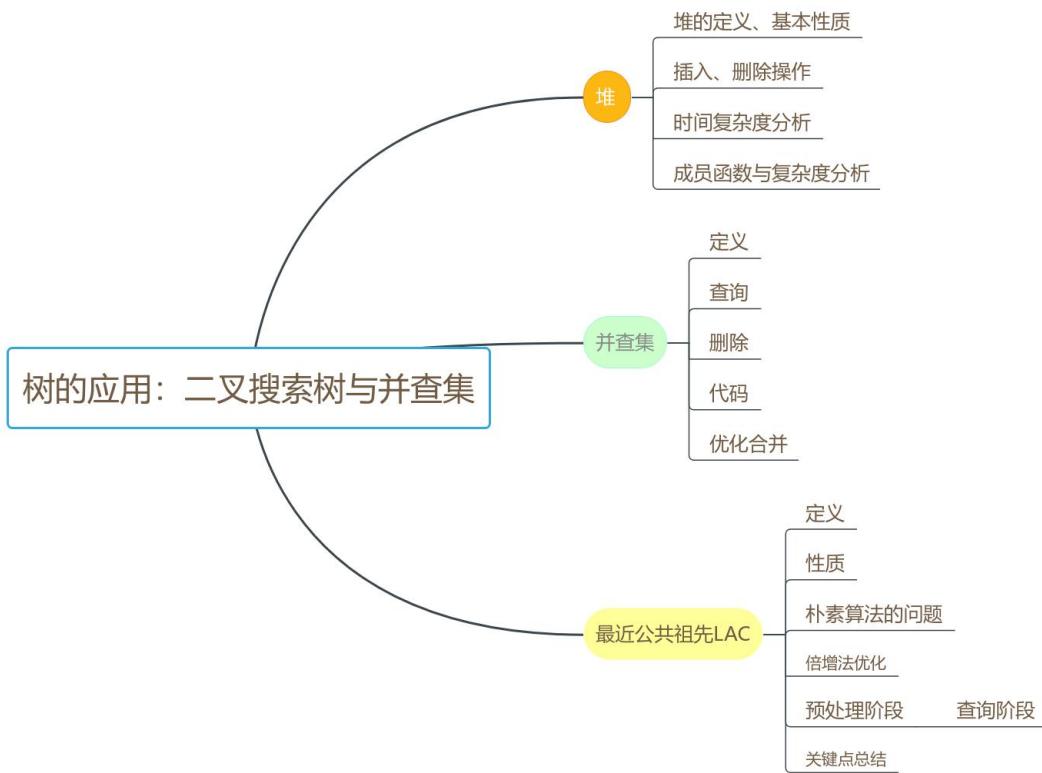
2. 扩展代码支持计算树上任意两点距离，并处理 m 次查询。

三、拓展题

1. 动态树场景：若允许树的边权动态修改，如何优化距离计算？

2. 多结点 LCA：如何高效计算三个结点 u, v, w 的 LCA？

1. 下图展示了本章的核心概念与关键能力，请同学们对照图中的内容进行总结。



2. 根据自己的掌握情况填写下表。

学习内容	掌握程度
二叉树的应用——堆	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
堆的定义、插入删除、复杂度	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
森林的应用——并查集的定义、操作	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
并查集的代码实现	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
最近公共祖先的定义、性质	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
最近公共祖先的预处理等阶段	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解

小结

在本章中，我们围绕树这一核心主题展开了深入探索。通过剖析树形结构的层次关系与递归特性，我们不仅掌握了二叉树的前序、中序、后序遍历等基础操作，更通过竞赛真题理解了完全二叉树、满二叉树等特殊结构的应用场景。在字典树的专题中，我们通过字符串匹配、前缀统计等经典问题，领略了这种高效数据结构在信息检索中的独特优势。本章内容着重培养从问题特征出发选择数据结构的能力——面对需要体现层次关系、快速前缀匹配或递归求解的问题时，树结构往往能提供更优雅的解决方案。

二叉搜索树（BST）是一种特殊的二叉树，具有以下性质：对于树中的每个节点，其左子树上所有节点的值均小于该节点的值，而右子树上所有节点的值均大于该节点的值。这种结构使得二叉搜索树在数据存储和检索方面表现出色，尤其是在动态数据集合中，能够高效地完成插入、删除和查找操作。例如，在实现字典、符号表等数据结构时，二叉搜索树可以快速定位目标元素，同时保持数据的有序性。

并查集是一种用于处理不相交集合的数据结构，支持两种基本操作：查找和合并。查找操作用于确定某个元素所属的集合，而合并操作则将两个不同的集合合并为一个。并查集的实现通常通过路径压缩和按秩合并来优化性能，使得查找和合并操作的时间复杂度接近常数。并查集在图论、网络设计以及动态连通性问题中有着广泛的应用。例如，在社交网络中，可以利用并查集快速判断两个用户是否属于同一个社交圈，或者在网络中判断两个节点是否连通。

本章内容还强调了二叉树的递归思维与并查集的空间优化技巧。在面对搜索路径优化、字符串处理、最优决策树构建等复杂问题时，这些技巧能够帮助我们更高效地设计算法。通过本章的学习，希望同学们能够将这些知识融会贯通，在后续高阶数据结构的学习中展现出更扎实的算法设计与问题拆解能力。

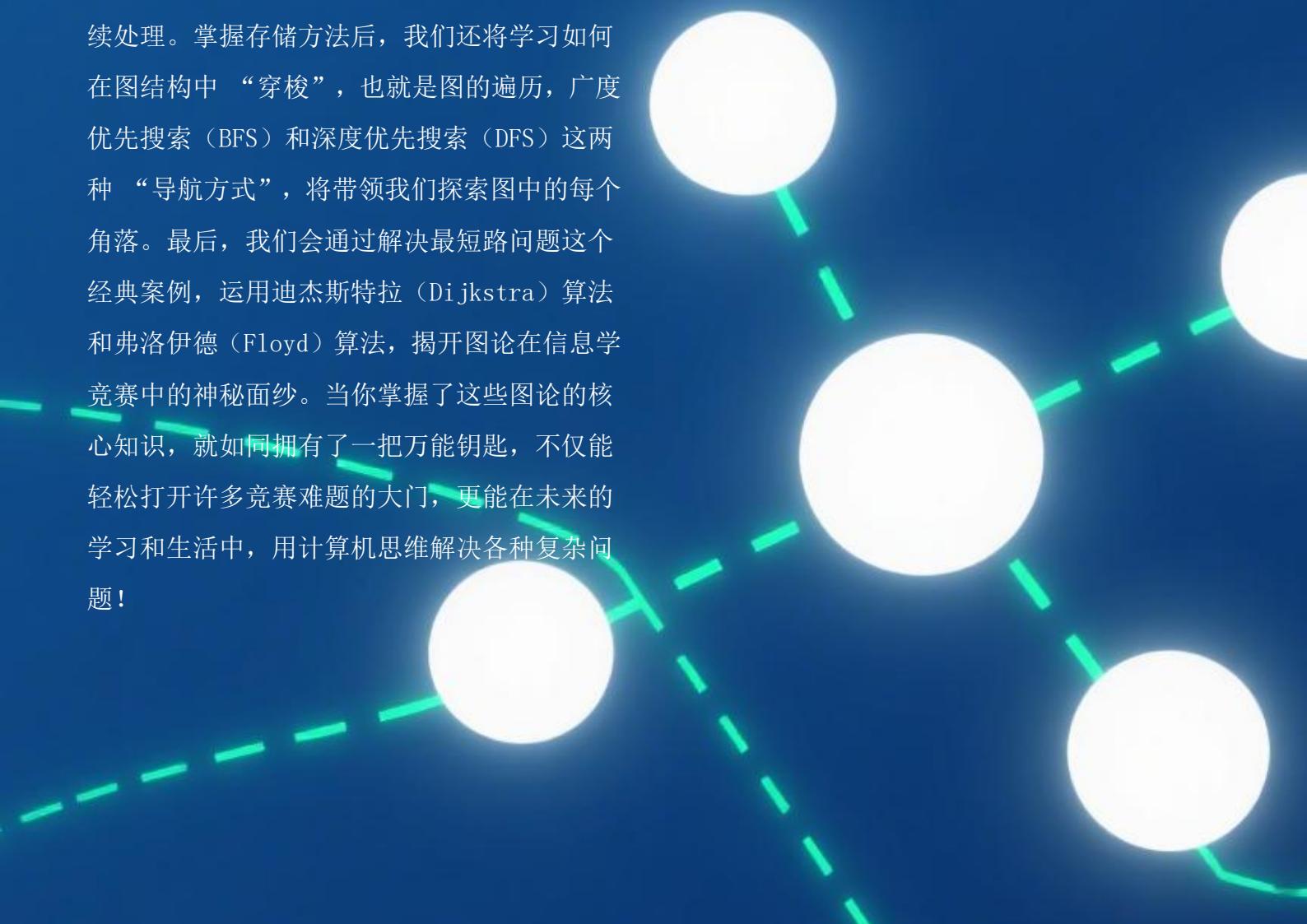
第四章

图论初步：存储与遍历

同学们，你是否曾好奇，外卖平台是如何在错综复杂的城市道路中，规划出骑手送餐的高效路线？网络游戏里，角色之间的社交关系网络又是怎样被构建和管理的？其实，这些生活中常见又有趣的现象背后，都藏着一个强大而实用的数学工具——图论。在此之前，我们已经认识了数组、链表、树等数据结构，它们能帮助我们处理很多问题，但在面对更复杂的多对多关系时，就需要“终极进化版”的数据结构——图结构登场了。

无论是像蜘蛛网般交织的城市间交通网络，每个城市作为一个顶点，连接城市的公路、铁路就是边；还是互联网中无数网页构成的链接世界，网页是顶点，超链接则是边，都可以抽象为由顶点和边组成的“图”。而我们即将开启的这一章节，就像是一场探索图论奥秘的奇妙之旅。

在这段旅程中，我们首先会深入了解图的基本概念，认识顶点、边、权值这些构成图的“基石”，区分无向图、有向图等不同类型图的特点。接着，我们要学习图在计算机中的存储方式，邻接矩阵和邻接表就像两种不同的“收纳方法”，能帮助我们高效地把图的数据存储起来，以便后续处理。掌握存储方法后，我们还将学习如何在图结构中“穿梭”，也就是图的遍历，广度优先搜索（BFS）和深度优先搜索（DFS）这两种“导航方式”，将带领我们探索图中的每个角落。最后，我们会通过解决最短路问题这个经典案例，运用迪杰斯特拉（Dijkstra）算法和弗洛伊德（Floyd）算法，揭开图论在信息学竞赛中的神秘面纱。当你掌握了这些图论的核心知识，就如同拥有了万能钥匙，不仅能轻松打开许多竞赛难题的大门，更能再未来的学习和生活中，用计算机思维解决各种复杂问题！



4.1 图的基本概念

本节学习目标

- ◆ 理解图的定义与相关要素的基本概念，能够掌握图的结构。
- ◆ 掌握图的基本术语（如顶点、边、有向边、无向边、权等）及其实际意义。
- ◆ 识别多种图的类型（如连通图、强连通图等）的结构特点。
- ◆ 掌握图的不同路径（如欧拉路径、哈密顿路径等），并能够说明其定义和区别。
- ◆ 通过学习提升对图这一非线性结构的抽象思维与实际问题建模能力。

4.1.1 图的组成

图的定义

图(**Graph**)是一种用于表示对象之间关系的非线性数据结构，由顶点(**Vertex/Node**)和边(**Edge**)组成。顶点通常表示实体或对象，而边则表示这些实体或对象之间的关系。接下来我们将带领大家学习表示图基本结构的各种术语以及不同种类的边、图，一起来探索图的奥秘！

如图4.4.1，即为一个基本的图。ABCDEF为顶点，连接顶点的就是边。

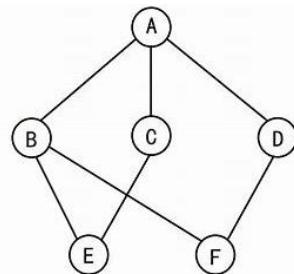


图4.1.1 基本的图

图的基本结构

1. 图由顶点和边组成：

(1) 顶点 (Vertex/Node)：顶点是图中的基本单元，表示图中的一个对象或实体。在图中，顶点通常用圆圈表示，并在圆圈内标注顶点的名称或编号。图4.4.1中圆圈A、B、C、D、E、F即为顶点。

(2) 边 (Edge)：边是连接两个顶点的连线，用于表示顶点之间的关系。

2. 边可以分为五种类型：

(1) 无向边 (Undirected Edge)：没有方向的边，表示两个顶点之间的双向连接。例如，无向边 (u,v) 表示顶点 u 和顶点 v 之间存在连接，且这种连接是双向的。图4.4.1中连接圆圈的没有箭头表示方向的连线即为无向边。

(2) 有向边 (Directed Edge)：带有方向的边，表示从一个顶点指向另一个顶点的单向连接。例如，有向边 $u \rightarrow v$ 表示从顶点 u 指向顶点 v 的连接。如图4.4.2中带有箭头的边。

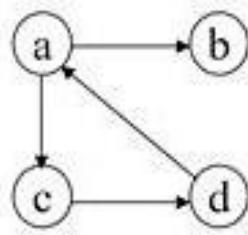


图4.1.2 有向边

(3) 多重边 (Multiple Edges)：两个顶点间存在多条边。边的数量用重表示。如图4.1.3中 v_1 和 v_2 两个顶点之间有两条边。

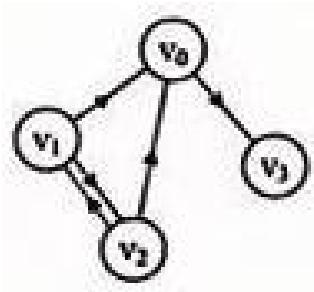


图4.1.3 多重边

(4) 自环边 (Loop)：顶点到自身的边。如图4.4.4中顶点0上的自环边。

(5) 平行边 (Parallel Edges)：两个顶点之间有两条无向边或有两条起点和终点相同的有向边。

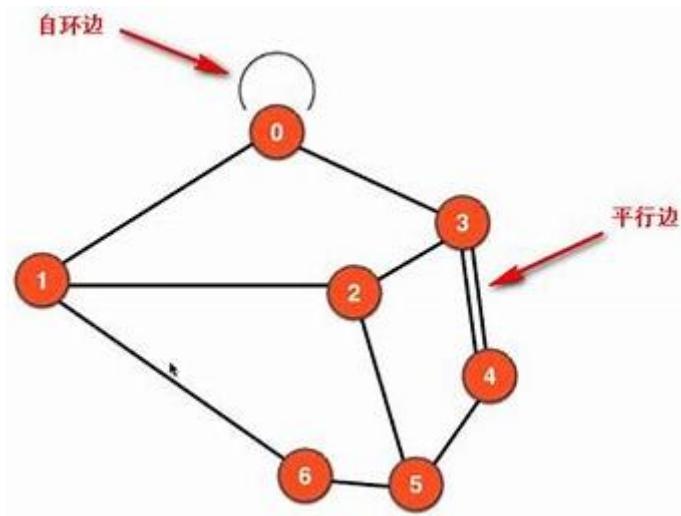


图4.1.4 复杂的图

4.1.2 图的分类

根据边的类型和方向分类

- 1. 无向图 (Undirected Graph) :** 图中的所有边都是无向边, 即边没有方向, 顶点之间的连接是双向的。图4.4.1就是无向图。
- 2. 有向图 (Directed Graph) :** 图中的所有边都是有向边, 即边有方向, 顶点之间的连接是单向的。图4.4.5就是有向图。大家注意进行区分, 关注不同图的关键特征。

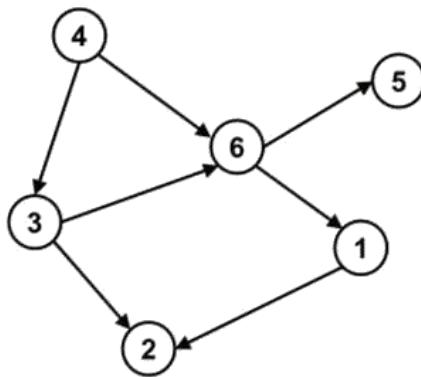


图4.1.5 有向图

- 3. 混合图 (Mixed Graph) :** 图中同时包含有向边和无向边。这种图在某些复杂的应用场景中可能会出现, 例如在交通网络中, 某些路段可能是双向通行的(无向边), 而某些路段可能是单向通行的(有向边)。

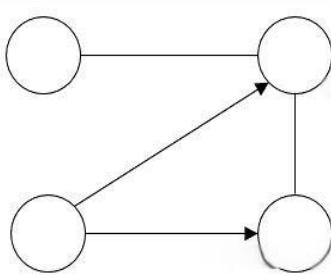


图4.1.6 混合图

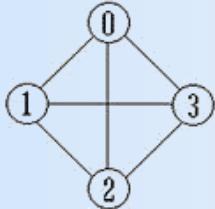


拓展知识：特殊类型的图

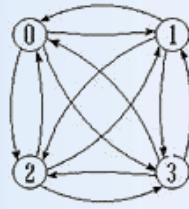
完全图

在无向图中，每对顶点之间都有一条边相连；在有向图中，每对顶点之间都有一条双向边相连，这样的图被称为完全图（Complete Graph）。

思考：当完全图的顶点数为 n 时，边数为多少呢？相信大家都计算出来了。在无向图中有 $n(n-1)/2$ 条边，在有向图中则有 $n(n-1)$ 条边。



无向完全图



有向完全图

图4.1.7 完全图

顶点的度数

1. 度数 (Degree) : 仅在无向图中适用，无向图顶点的度数是指与该顶点相连的边的数量。例如，如果一个顶点与三条边相连，那么它的度数为 3。

2. 入度 (In-Degree) 和出度 (Out-Degree) : 仅在有向图中适用，有向图中入度是指指向该顶点的边的数量，出度是指从该顶点出发的边的数量。例如，对于顶点 v ，如果有三条边指向 v ，而从 v 出发的边有两条，那么 v 的入度为 3，出度为 2。

4.1.3 图的路径与连通性

在图的理论中，路径和连通性是两个非常重要的概念。它们不仅描述了图中顶点之间的连接关系，还为许多图算法提供了基础。本小节将详细介绍这些概念及其相关术语。

连通性

连通性（Connectivity） 是图的一个基本性质，用于描述图中顶点之间的可达性。

1. 连通图（Connected Graph）：在无向图中，如果任意两个顶点之间都存在路径，则称该图为连通图。路径是指从一个顶点到另一个顶点的顶点序列，序列中的相邻顶点通过边相连。

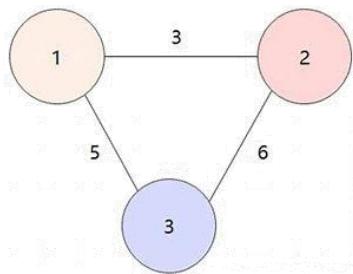


图4.1.8 连通图

2. 强连通图（Strongly Connected Graph）：在有向图中，如果任意两个顶点u和v之间都存在从u到v和从v到u的路径，则称该图为强连通图。这意味着在强连通图中，任意两个顶点之间都可以双向到达。

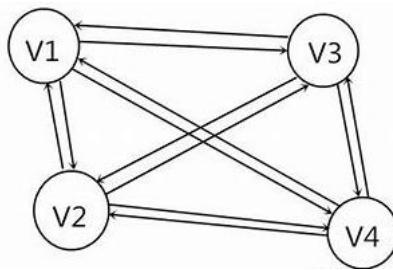


图4.1.9 强连通图

路径与回路

路径和回路是图中描述顶点之间连接关系的重要概念。

1. 路径 (Path) : 路径是从一个顶点到另一个顶点的顶点序列，序列中的相邻顶点通过边相连。路径的长度是指路径中边的数量。例如，在图4.1.10的图中，路径 $B \rightarrow D \rightarrow F$ 表示从顶点 B 经过顶点 D 到达顶点 F 的一条路径。

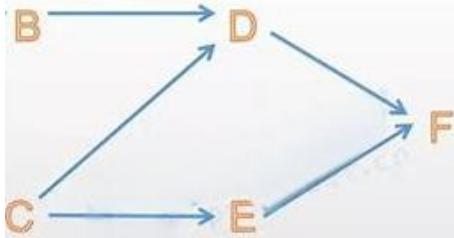


图4.1.10 路径示意图

2. 简单路径 (Simple Path) : 简单路径是指路径中顶点不重复的路径。也就是说，路径中的每个顶点（除了起点和终点）都只出现一次。例如，路径 $B \rightarrow D \rightarrow F$ 是简单路径，而路径 $B \rightarrow D \rightarrow B \rightarrow D \rightarrow F$ 不是简单路径，因为顶点 B 和 D 重复出现了。

3. 回路 (Cycle) : 回路是指起终点相同的路径。也就是说，路径的起点和终点是同一个顶点。例如，路径 $v_0 \rightarrow v_2 \rightarrow v_3 \rightarrow v_0$ 是一个回路。

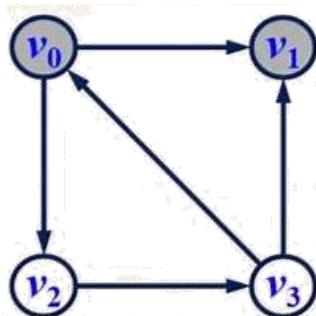


图4.1.11 回路示意图

4. 简单回路 (Simple Cycle) : 简单回路是指除起点和终点外，路径中顶点不重复的回路。例如在图4.1.11中，路径 $v_0 \rightarrow v_2 \rightarrow v_3 \rightarrow v_0$ 是简单回路，而路径 $v_0 \rightarrow v_2 \rightarrow v_3 \rightarrow v_0 \rightarrow v_2 \rightarrow v_3 \rightarrow v_0$ 不是简单回路，因为顶点 v_2 和 v_3 重复出现了。

特殊路径

1. 欧拉路径 (Eulerian Path) : 欧拉路径是指在图中遍历每条边恰好一次的路径。如果一个图存在欧拉路径，那么这个图称为欧拉图。欧拉路径的起点和终点可以不同。

图4.1.12中每条边上的序号为遍历的顺序，在图中遍历每条边恰好一次。

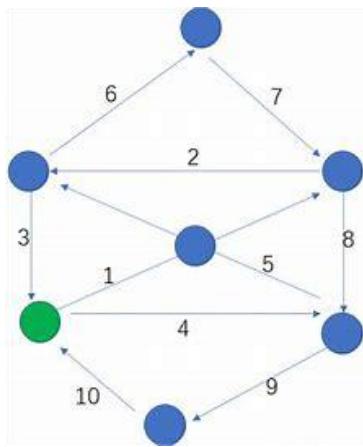


图4.1.12 欧拉图

2. 哈密顿路径（Hamiltonian Path）：哈密顿路径是指在图中遍历每个顶点恰好一次的路径。如果一个图存在哈密顿路径，那么这个图称为哈密顿图。哈密顿路径的起点和终点可以不同。

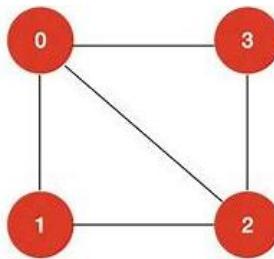


图4.1.13 哈密顿图



知识链接：边权重

边权重是边的数值属性，通常用于表示边的某种特性，用于最短路径或最小生成树。请大家回忆之前的知识，想一想如何运用呢？

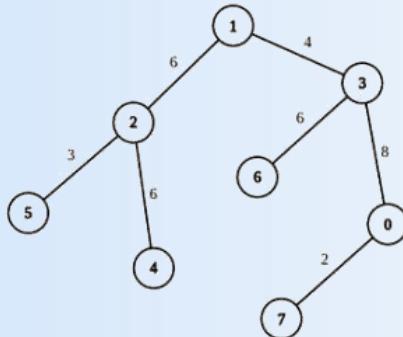
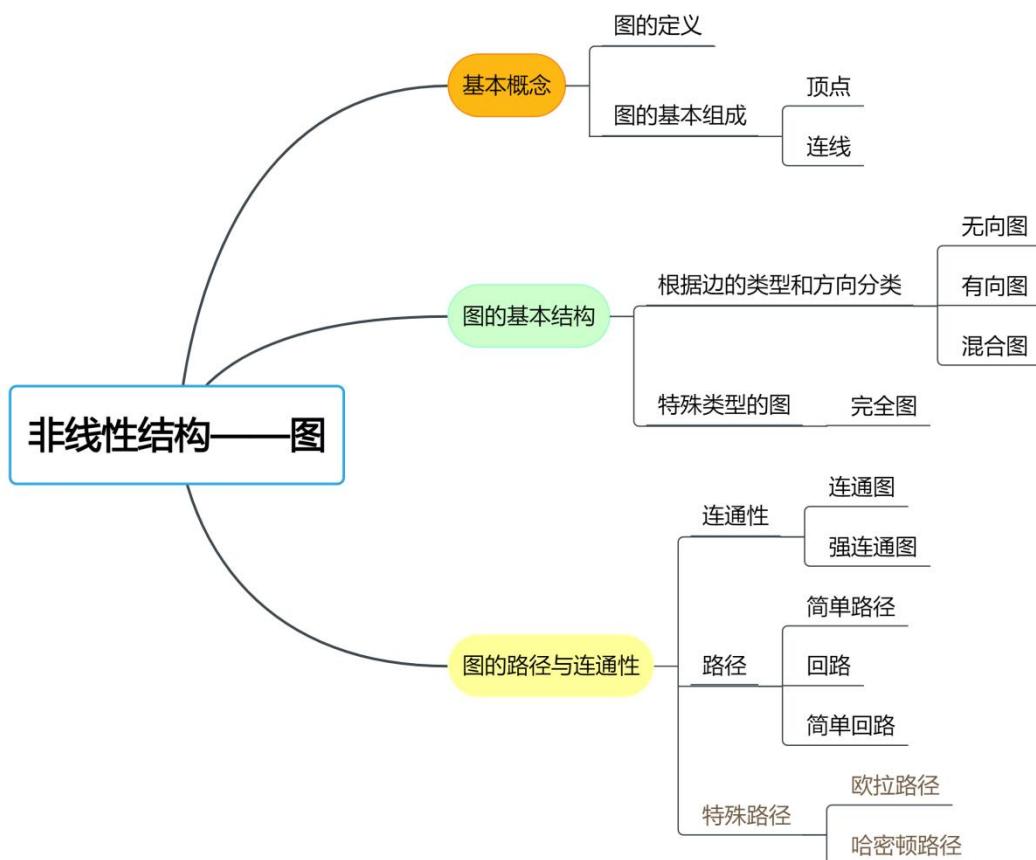


图4.1.14 边权重



知识图谱



练习提升

一、判断题

1. 有向图中，如果从顶点 u 到顶点 v 有路径，那么从顶点 v 到顶点 u 一定有路径。 ()
2. 无向图中，如果两个顶点之间有边，那么这两个顶点一定连通。 ()
3. 欧拉路径是指在图中遍历每条边恰好一次的路径。 ()
4. 哈密顿路径是指在图中遍历每条边恰好一次的路径。 ()

二、选择题

1. 下列关于图的说法中，正确的是 ()。

- A. 有向图中，如果从顶点 u 到顶点 v 有路径，那么从顶点 v 到顶点 u 一定有路径。
- B. 无向图中，如果两个顶点之间有边，那么这两个顶点一定连通。
- C. 有向图中，如果从顶点 u 到顶点 v 有路径，那么从顶点 v 到顶点 u 一定没有路径。
- D. 无向图中，如果两个顶点之间没有边，那么这两个顶点一定不连通。
2. 下列关于图的路径的说法中，错误的是（ ）。
- A. 简单路径是指路径中顶点重复的路径。
- B. 回路是指起终点相同的路径。
- C. 欧拉路径是指在图中遍历每条边恰好一次的路径。
- D. 哈密顿路径是指在图中遍历每个顶点恰好一次的路径。

三、计算题

- 给定一个无向图，顶点集合为 $V=\{A,B,C,D,E\}$ ，边集合为 $E=\{(A,B),(A,C),(B,C),(B,D),(C,D),(D,E)\}$ 。判断该图是否为连通图。
- 给定一个有向图，顶点集合为 $V=\{A,B,C,D\}$ ，边集合为 $E=\{(A,B),(B,C),(C,D),(D,A),(A,C)\}$ 。判断该图是否为强连通图。

四、应用题

- 在一个交通网络中，有5个城市，分别用顶点 A,B,C,D,E 表示。城市之间的道路用边表示，道路的长度用边的权重表示。给定边集合和权重如下：
 $E=\{(A,B,10),(A,C,15),(B,C,20),(B,D,25),(C,D,30),(D,E,35)\}$ 。

问题：从城市 A 到城市 E 的最短路径是什么？

- 在一个社交网络中，有4个用户，分别用顶点 A,B,C,D 表示。用户之间的关系用边表示，给定边集合如下：

$$E=\{(A,B),(A,C),(B,C),(B,D),(C,D)\}.$$

问题：该社交网络是否为连通图？为什么？

4.2 图的存储与遍历

本节学习目标

- ◆ 理解图的四种存储结构（边集、邻接表、邻接矩阵、链式前向星）的实现原理与应用场景。
- ◆ 掌握邻接表与邻接矩阵的核心操作时间复杂度对比，能针对“判断边存在性”“遍历邻接点”等操作，结合图结构特点选择最优存储方式。
- ◆ 熟练运用BFS和DFS算法完成图的遍历，能用队列实现BFS、递归/栈实现DFS，理解二者在最短路径、连通性问题中的应用差异。

4.2.1 图的存储方法

边集存储

边集存储（Edge List）是最直观、最基础的存储方式，它将图中的每一条边都单独记录下来。对于一个拥有E条边的图，可使用数组edges[E][2]（适用于无向图）来存储边信息，其中edges[i][0]和edges[i][1]分别表示第i条边连接的两个顶点。以一个简单无向图为例：

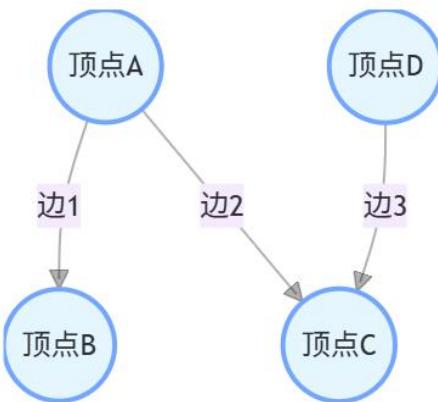


图4.2.1 简单的无向图

在上述图中，若采用边集存储，对应的数组内容为[[A, B], [A, C], [D, C]]。

尽管边集存储方式简单直接，但在实际竞赛应用中却并不常用。这是因为当我们需要查询某个顶点的所有相邻顶点时，需要遍历整个边集数组，时间复杂度高达 $O(E)$ 。例如，若要查找顶点A的邻接点，需逐个检查数组中的每一条边，效率较低，因此通常仅作为理解图存储的基础概念。

邻接表存储

邻接表（Adjacency List）是竞赛中广泛使用的存储方式，它使用`vector<vector<int>>`来存储边信息。形象地说，邻接表为图中的每个顶点建立了一个“邻居清单”。外层容器对应图中的各个顶点，内层容器则记录与该顶点直接相连的其他顶点。

例如对于顶点A，其内层容器会存储所有与A有边相连的顶点。

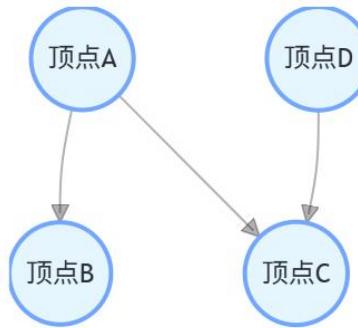


图4.2.2 邻接表示意图

以图4.2.2为例，使用 C++ 实现邻接表存储的代码如下：

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 const int N = 100; // 最大顶点数
6 vector<int> adj[N]; // 邻接表
7
8 void addEdge(int u, int v) {
9     adj[u].push_back(v);
10    adj[v].push_back(u); // 无向图，双向添加
11 }
```

图4.2.3 邻接表存储代码（上）

```

13 int main() {
14     addEdge(0, 1);
15     addEdge(0, 2);
16     addEdge(1, 2);
17
18     // 输出顶点0的邻接点
19     cout << "顶点0的邻接点: ";
20     for (int i = 0; i < adj[0].size(); i++) {
21         cout << adj[0][i] << " ";
22     }
23     return 0;
24 }

```

图4.2.4 邻接表存储代码（下）

邻接表存储方式在处理稀疏图（边数远小于顶点数平方的图）时具有显著优势，其空间复杂度为 $O(V + E)$ ，仅存储实际存在的边，避免了空间浪费。同时，遍历某个顶点的所有邻接点时，只需访问其对应的内层容器，时间复杂度为 $O(E)$ ，效率较高。

邻接矩阵存储

邻接矩阵（Adjacency Matrix）采用二维数组来存储图结构。假设图中有 v 个顶点，则创建一个二维数组 $graph[V][V]$ 。对于无向图，若顶点*i*和顶点*j*之间存在边，则将 $graph[i][j]$ 和 $graph[j][i]$ 设为1（也可设为其他表示边存在的值）；若不存在边，则设为0。例如，对于一个包含3个顶点的无向图，若顶点0和顶点1有边相连，其邻接矩阵如下：

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

图4.2.5 邻接矩阵

使用 C++ 实现邻接矩阵存储的代码如下：

```

1 #include <iostream>
2 using namespace std;
3
4 const int N = 100; // 最大顶点数
5 int graph[N][N]; // 邻接矩阵
6
7 void addEdge(int u, int v) {
8     graph[u][v] = 1;
9     graph[v][u] = 1; // 无向图, 双向标记
10 }
11
12 int main() {
13     addEdge(0, 1);
14     addEdge(0, 2);
15     addEdge(1, 2);
16
17     // 输出邻接矩阵
18     for (int i = 0; i < 3; i++) {
19         for (int j = 0; j < 3; j++) {
20             cout << graph[i][j] << " ";
21         }
22         cout << endl;
23     }
24     return 0;
25 }

```

图4.2.6 邻接矩阵存储代码

邻接矩阵的最大优点在于判断两点之间是否存在边极为快速，只需直接访问对应数组元素，时间复杂度为 $O(1)$ 。然而，其缺点也十分明显，无论图的稀疏程度如何，都需要占用 $O(VA^2)$ 的空间，当处理稀疏图时，会造成大量空间浪费。

链式前向星

链式前向星（Linked Forward Star） 是一种基于链表思想、通过数组模拟链表实现的邻接表变体，常用于高效存储和处理图结构，尤其在需要对边进行特殊操作时表现出色。

在链式前向星中，通常需要定义以下几个关键数组：

- **head[i]**: 表示以顶点*i*为起点的第一条边在边数组edge中的存储位置。

- **edge[i].to:** 表示第(i)条边的终点。
- **edge[i].next:** 表示与第(i)条边同起点的下一条边在边数组edge中的存储位置。
- **cnt:** 用于记录当前边的数量。

以下是一个简单的链式前向星存储的 C++ 代码示例：

```

1 #include <iostream>
2 using namespace std;
3
4 const int N = 100, M = 1000; // 最大顶点数和最大边数
5 int head[N], cnt;
6 struct Edge {
7     int to, next;
8 } edge[M];
9
10 void addEdge(int u, int v) {
11     edge[cnt].to = v;
12     edge[cnt].next = head[u];
13     head[u] = cnt++;
14 }
15
16 int main() {
17     addEdge(0, 1);
18     addEdge(0, 2);
19     addEdge(1, 2);
20
21     // 遍历顶点0的邻接边
22     for (int i = head[0]; i != -1; i = edge[i].next) {
23         int v = edge[i].to;
24         cout << v << " ";
25     }
26     return 0;
27 }
```

图4.2.7 链式前向星存储代码

还是以图二为例，其链式前向星存储结构可通过如下图示展示。在该图示中，顶点A有两条出边，head[A]指向第一条边E1（在edge数组中位置为0），E1的next指向同起点的第二条边E2（位置为1），E2的next为-1表示无后续同起点边；顶点D的出边情况同理。通过这种结构，能够高效地遍历每个顶点的邻接边。

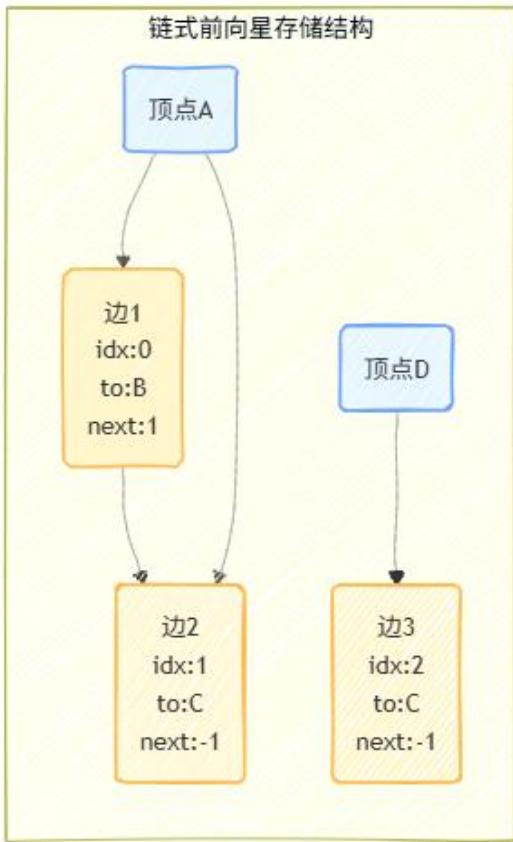


图4.2.8 链式前向星存储结构示意图

4.2.2 几种存储方法的比较

为了更全面、深入地理解不同图存储方法的特性，我们从存储结构、时间复杂度、空间复杂度和优缺点等多个维度进行详细对比。

存储结构对比

方法	实现	结构特点
邻接表	vector (C++)	为每个顶点构建邻接顶点列表，动态存储实际存在的边
邻接矩阵	二维数组	使用固定大小的方阵，通过矩阵元素标记顶点间连接关系
边集	线性列表或数组	简单记录图中每一条边的两个端点
链式前向星	多个数组 (head数组、edge结构体数组等)	通过数组模拟链表，存储边的详细信息及连接关系

表4-1 存储结构对比表

时间复杂度对比

(注意: E指边数, V指节点数)

操作	邻接表	邻接矩阵	边集	链式前向星	分析
判断边存在性	$O(E)$	$O(1)$	$O(E)$	$O(E)$	邻接表和链式前向星需遍历邻接边查找; 邻接矩阵直接访问对应元素; 边集需遍历所有边判断
遍历邻接点	$O(E)$	$O(V)$	$O(E)$	$O(E)$	邻接表和链式前向星遍历邻接边; 邻接矩阵遍历对应行; 边集需遍历所有边筛选
添加边	$O(1)$	$O(1)$	$O(1)$	$O(1)$	均为简单的插入或标记操作

表4-2 时间复杂度对比表

空间复杂度对比

方法	空间复杂度	说明	适用场景
邻接表	$O(V + E)$	仅存储实际边和顶点信息, 动态分配	稀疏图
邻接矩阵	$O(V^2)$	固定大小数组, 无论图疏密均占用相同空间	稠密图
边集	$O(E)$	仅存储边信息, 不记录顶点间接关系	需全局边操作场景
链式前向星	$O(V + E)$	与邻接表类似, 但需额外存储边连接关系	需高效边操作场景

表4-3 空间复杂度对比表

优缺点对比

方法	优点	缺点
邻接表	空间利用率高, 适合稀疏图; 遍历邻接点效率高; 动态存储灵活适应图规模变化	动态分配存在一定开销; 判断边存在性需遍历邻接边, 效率较低
邻接矩阵	判断边存在性极为快速; 矩阵操作方便, 易于实现	空间浪费严重, 尤其在稀疏图中; 占用固定空间, 无法动态调整

边集	结构简单，易于实现；适合对全局边进行统一操作，如排序	遍历邻接点需遍历整个边集，效率低下；无法快速获取顶点邻接信息
链式前向星	空间利用率高，存储紧凑；适合对边进行特殊操作和处理	原理相对复杂，实现难度较高；代码调试和维护成本较大

表4-4 优缺点对比表

4.2.3 图的遍历

完成图的存储后，图的遍历操作是解决众多图论问题的核心环节。其中，广度优先搜索（BFS）和深度优先搜索（DFS）是最常用的两种遍历策略。下面以常用的邻接表存储为例，详细介绍这两种遍历方式。

广度优先搜索（BFS）

BFS 的遍历过程如同水波扩散，从起始顶点开始，逐层向外访问顶点。首先访问起始顶点的所有直接邻居，然后依次访问这些邻居的未访问邻居，以此类推，直到遍历完所有可达顶点。在实现 BFS 时，通常借助队列来管理待访问的顶点，这是因为队列先进先出的特性，恰好符合 BFS 逐层探索的逻辑。以下简单图为例：

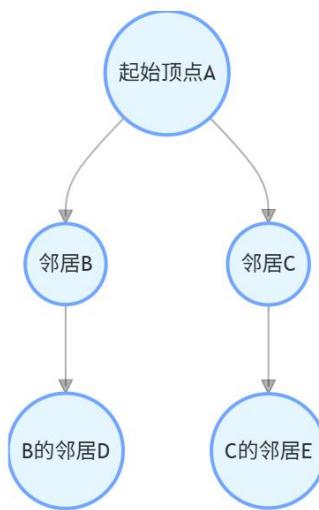


图4.2.9 广度优先搜索示意图

BFS 在很多场景中都有重要应用。例如，在寻找无权图中两个顶点的最短路径时，BFS 能够保证找到的路径是经过边数最少的路径；在求解迷宫问题时，BFS 可以按照离起点由近到远的顺序探索迷宫，找到从起点到终点的最短路线。

使用 C++ 实现 BFS 的代码如下：

```
1 #include <iostream>
2 #include <vector>
3 #include <queue>
4 using namespace std;
5
6 const int N = 100;
7 vector<int> adj[N];
8 bool visited[N];
9
10 void bfs(int start) {
11     queue<int> q;
12     q.push(start);
13     visited[start] = true;
14
15     while (!q.empty()) {
16         int u = q.front();
17         q.pop();
18         cout << u << " ";
19
20         for (int i = 0; i < adj[u].size(); i++) {
21             int v = adj[u][i];
22             if (!visited[v]) {
23                 q.push(v);
24                 visited[v] = true;
25             }
26         }
27     }
28 }
```

图4.2.10 BFS代码

深度优先搜索（DFS）

DFS 的遍历方式更像是一场不断深入的冒险，从起始顶点开始，沿着一条路径尽可能地深入访问相邻顶点，直到无法继续前进（即当前顶点的所有邻接顶点都已被访问），然后回溯到上一个顶点，继续探索其他未访问的路径，直至遍历完所有可达顶点。DFS 可以使用递归或者栈来实现，递归实现代码简洁直观，而栈实现则更便于理解其底层逻辑。

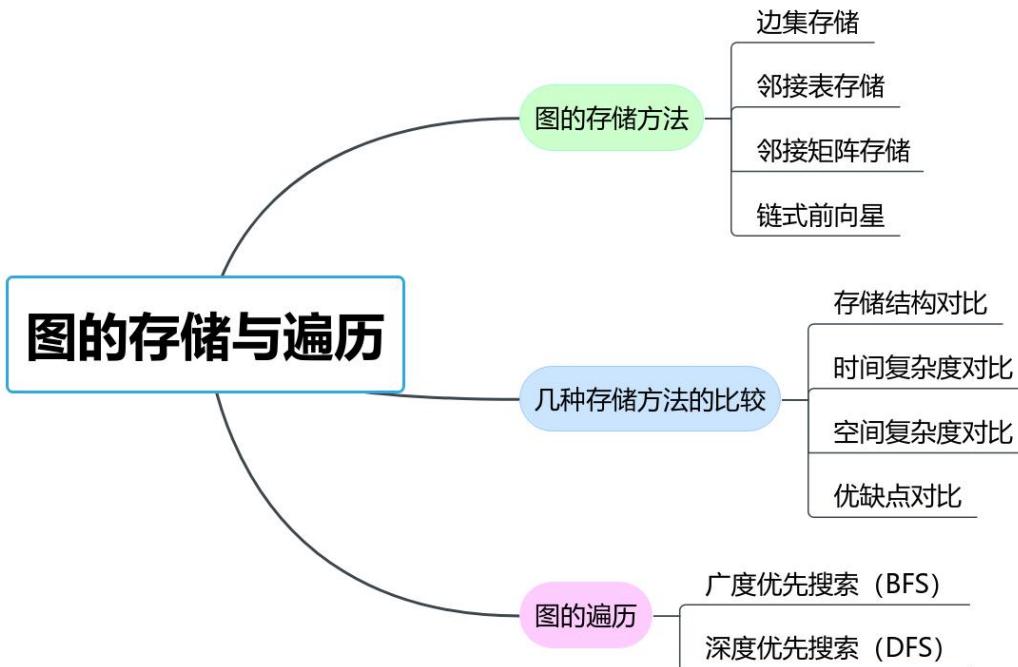
同样以之前的简单图为例，使用 C++ 递归实现 DFS 的代码如下：

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 const int N = 100; // 定义最大顶点数
6 vector<int> adj[N]; // 邻接表存储图结构
7 bool visited[N]; // 标记数组，用于记录顶点是否已被访问
8
9 void dfs(int u) {
10     visited[u] = true; // 标记当前顶点已访问
11     cout << u << " "; // 输出当前访问的顶点
12
13     for (int i = 0; i < adj[u].size(); i++) { // 遍历当前顶点的所有邻接顶点
14         int v = adj[u][i]; // 获取邻接顶点
15         if (!visited[v]) { // 如果邻接顶点未被访问
16             dfs(v); // 递归访问该邻接顶点
17         }
18     }
19 }
```

图4.2.11 DFS代码

DFS 在拓扑排序、寻找图的连通分量等问题中发挥着重要作用。例如，在有向无环图的拓扑排序中，通过 DFS 可以方便地确定顶点的先后顺序；在判断一个图是否连通时，从任意一个顶点开始进行 DFS，如果能够访问到所有顶点，则图是连通的，否则图存在多个连通分量。

通过对 BFS 和 DFS 的详细介绍，我们可以根据具体问题的需求和特点，选择合适的遍历方式，从而高效地解决各类图论问题。在后续的学习中，我们还会结合更多实际竞赛题目，深入探讨它们的应用技巧和优化方法。



一、基础概念题

1. 比较邻接矩阵和邻接表的优缺点，并分别说明它们适用的场景（如稠密图、稀疏图）。
2. 给定一个图，顶点数为V，边数为E。请填写以下操作在不同存储方式下的时间复杂度：

操作	邻接表	邻接矩阵	边集	链式前向星
判断顶点u和v是否邻接				
遍历顶点u的所有邻接点				
添加一条边u→v				

二、代码实现题

3. 邻接表的BFS遍历

使用C++实现邻接表存储的无向图，并从顶点0出发进行BFS遍历，输出访问顺序。

输入示例：顶点数 $V=4$ ，边集为 $\{\{0,1\}, \{0,2\}, \{2,3\}\}$ 。

输出示例：0 1 2 3

4. 链式前向星的DFS遍历

使用链式前向星存储有向图，并实现非递归（栈）的DFS遍历，从顶点0出发输出访问顺序。

输入示例：顶点数 $V=3$ ，边集为 $\{\{0,1\}, \{0,2\}, \{2,1\}\}$ 。

输出示例：0 2 1（依赖遍历顺序）

三、综合应用题

5. 最短路径问题

给定一个无权无向图（边权为1），使用邻接表存储，编写BFS算法求解从顶点 s 到顶点 t 的最短路径长度。若不可达返回-1。

输入示例：

$V=5, E=6, edges=\{\{0,1\}, \{0,2\}, \{1,3\}, \{2,3\}, \{2,4\}, \{3,4\}\}$

$s=0, t=4$

输出：3（路径 $0 \rightarrow 2 \rightarrow 4$ ）

6. 图的连通分量计数

使用邻接矩阵存储图，通过DFS遍历统计图中的连通分量数量。

输入示例：

$V=5, edges=\{\{0,1\}, \{1,2\}, \{3,4\}\}$ // 图分为两个连通分量

输出：2

四、进阶思考题

7. 存储方式选择与优化

假设需要频繁进行两种操作：(1) 判断任意两项点是否邻接；(2) 遍历某个顶点的所有邻接点。

若图的顶点数 $V=1000$ ，边数 $E=3000$ ，你会选择哪种存储方式？为什么？如果边数变为 $E=800000$ ，选择会变化吗？

4.3 图上问题案例——最短路

本节学习目标

- ◆ 理解Dijkstra算法、Bellman - Ford算法、Floyd - Warshall算法这三种算法解决图的最短路径问题的核心原理，熟知各算法的特性。
- ◆ 能够独立用 C++实现三种算法，学会调试代码以解决可能出现的逻辑错误、运行错误，了解算法性能优化的方向理解并查集的时间复杂度。
- ◆ 通过学习这三种算法，培养贪心、动态规划的算法思维，遇到问题能选择合适策略，学会分析算法的时间和空间复杂度。

4.3.1 单源最短路径算法——Dijkstra算法

算法简介

Dijkstra算法是由荷兰计算机科学家狄克斯特拉于1959年提出的，因此又叫狄杰斯特拉算法。是从一个顶点到其余各顶点的最短路径算法，解决的是有权图中最短路径问题。该算法主要特点是从起始点开始，采用贪心算法的策略，每次遍历到始点距离最近且未访问过的顶点的邻接节点，直到扩展到终点为止。普通的 Dijkstra 算法在处理大规模图时，由于每次都需要遍历所有未确定最短路径的顶点来找到距离源点最近的顶点，时间复杂度较高，为 $O(V^2)$ ，其中 V 是图中顶点的数量。在竞赛中，当图的规模较大时，这种复杂度可能会导致程序超时。因此，我们需要一种更高效的方法来优化 Dijkstra 算法，这就是基于堆优化的 Dijkstra 算法。

而基于堆优化的 Dijkstra 算法引入了优先队列（通常使用小顶堆实现）来优化选择最近顶点的过程。其核心思想是使用小顶堆来代替普通 Dijkstra 算法中的线性查找，从而将每次查找距离源点最近的顶点的时间复杂度从 $O(V)$ 降低到 $O(\log V)$ 。

算法详解

1. 初始化

创建一个数组 dist , 用于记录源点到每个顶点的最短距离。将源点的距离初始化为 0, 即 $\text{dist}[s] = 0$, 其他顶点的距离初始化为无穷大。

创建一个布尔类型的数组 visited , 用于标记每个顶点是否已经确定了最短路径。初始时, 所有顶点的标记都设为 false。

创建一个小顶堆 pq , 把源点及其距离 (0) 作为一个元素插入到堆中。堆中的元素通常是一个二元组 ($\text{distance}, \text{vertex}$), 按照距离从小到大排序。

2. 迭代过程

从优先队列 pq 中取出距离最小的顶点 u 。若该顶点已被标记为已访问 (即 $\text{visited}[u] == \text{true}$) , 则跳过该顶点, 继续从堆中取出下一个元素; 否则, 将该顶点标记为已访问, 即 $\text{visited}[u] = \text{true}$ 。

对于顶点 u 的所有邻接顶点 v , 计算通过 u 到达 v 的新距离 $\text{new_dist} = \text{dist}[u] + \text{weight}(u, v)$, 其中 $\text{weight}(u, v)$ 是边 (u, v) 的权值。若 new_dist 小于当前记录的 $\text{dist}[v]$, 则更新 $\text{dist}[v] = \text{new_dist}$, 并将 $(\text{new_dist}, v)$ 插入到优先队列 pq 中。

3. 终止条件

当优先队列为空时, 算法结束, 此时 dist 数组中存储的就是源点到每个顶点的最短距离。

接下来我们通过一道例题来详细学习基于堆优化的 Dijkstra 算法。

例题分析

【题目】

给定一个 n 个点, m 条有向边的带非负权图, 请你计算从 s 出发, 到每个点的距离。数据保证你能从 s 出发到任意点。

【输入格式】

第一行为三个正整数 n, m, s 。第二行起 m 行, 每行三个非负整数 u_i, v_i, w_i , 表示从 u_i 到 v_i 有一条权值为 w_i 的有向边。

【输出格式】

输出一行 n 个空格分隔的非负整数，表示 s 到每个点的距离。

【分步解析】

1. 定义边的结构体和无穷大常量：Edge 结构体用于表示图中的边，包含两个成员变量：to 表示边的终点，weight 表示边的权重。INF 常量表示无穷大，用于初始化距离数组。代码如图4.3.1所示。

```
// 定义边的结构体
struct Edge {
    int to;
    int weight;
    Edge(int t, int w) : to(t), weight(w) {}
};

// 定义无穷大常量
const int INF = INT_MAX;
```

图4.3.1 定义边的结构体和无穷大常量

2. 初始化：dist 数组用于存储从源点到每个顶点的最短距离，初始时将所有顶点的距离设为无穷大，源点的距离设为 0。pq 是一个优先队列（小顶堆），用于存储（距离，顶点）对，每次从队列中取出距离最小的顶点进行处理。代码如图4.3.2所示。

```
// 初始化距离数组，将所有顶点的距离设为无穷大
vector<int> dist(n + 1, INF);
// 源点的距离设为 0
dist[s] = 0;

// 定义优先队列（小顶堆），存储（距离，顶点）对
priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>> pq;
// 将源点及其距离加入优先队列
pq.push({0, s});
```

图4.3.2 初始化代码

3. 迭代过程：当队列不为空时，主循环不断从优先队列中取出距离最小的顶点进行处理；当队列为空时，满足终止条件，算法结束，此时 dist 数组中存储的就是源点到每个顶点的最短距离。

代码如图4.3.3所示：

```

while (!pq.empty()) {
    // 取出当前距离最小的顶点
    int currentDist = pq.top().first;
    int currentVertex = pq.top().second;
    pq.pop();

    // 如果当前距离大于已记录的距离，跳过
    if (currentDist > dist[currentVertex]) continue;

    // 遍历当前顶点的所有邻接边
    for (const Edge& edge : graph[currentVertex]) {
        int neighbor = edge.to;
        int weight = edge.weight;
        // 计算通过当前顶点到达邻接顶点的新距离
        int newDist = dist[currentVertex] + weight;

        // 如果新距离小于已记录的距离，更新距离并加入优先队列
        if (newDist < dist[neighbor]) {
            dist[neighbor] = newDist;
            pq.push({newDist, neighbor});
        }
    }
}

```

图4.3.3 迭代过程代码

4. 主函数调用：主函数首先读取顶点数 n、边数 m 和源点 s。初始化图的邻接表 graph，并读取每条边的信息。调用 dijkstra 函数计算从源点到所有顶点的最短路径。输出结果，即从源点到每个顶点的最短距离。代码如图4.3.4所示。

```

int main() {
    int n, m, s;
    // 读取顶点数、边数和源点
    cin >> n >> m >> s;

    // 初始化图的邻接表
    vector<vector<Edge>> graph(n + 1);

    // 读取每条边的信息
    for (int i = 0; i < m; ++i) {
        int u, v, w;
        cin >> u >> v >> w;
        graph[u].emplace_back(v, w);
    }

    // 调用 Dijkstra 算法计算最短路径
    vector<int> dist = dijkstra(n, s, graph);

    // 输出结果
    for (int i = 1; i <= n; ++i) {
        cout << dist[i];
        if (i < n) cout << " ";
    }
    cout << endl;

    return 0;
}

```

图4.3.4 主函数调用代码

时间复杂度分析

基于堆优化的Dijkstra算法有两个阶段——初始化阶段和迭代阶段，接下来将详细说明这两个阶段的时间复杂度。

初始化阶段又分为距离数组初始化和优先队列初始化。距离数组初始化需要将源点到自身的距离设为 0，到其他顶点的距离设为无穷大。这个操作需要遍历所有的 V 个顶点，因此时间复杂度为 $O(V)$ 。优先队列初始化将源点及其距离 (0) 插入到优先队列中，插入操作在小顶堆中的时间复杂度为 $O(\log V)$ ，但由于只插入一个元素，所以这部分时间复杂度可近似看作常数时间 $O(1)$ 。综合来看，初始化阶段的时间复杂度主要由距离数组初始化决定，为 $O(V)$ 。

迭代阶段也分两阶段：取出最小元素、更新距离并插入元素。在每次迭代中，需要从优先队列中取出距离最小的顶点。优先队列的取出最小元素操作的时间复杂度为 $O(\log V)$ 。而整个算法最多需要进行 V 次取出操作（因为每个顶点最多被取出一次），所以这部分的总时间复杂度为 $O(V \log V)$ 。对于每个顶点，需要遍历其所有的邻接边来更新相邻顶点的距离。图中所有顶点的邻接边总数为 E 。当发现通过当前顶点到达某个相邻顶点的距离更短时，需要更新该相邻顶点的距离，并将其插入到优先队列中。插入操作在小顶堆中的时间复杂度为 $O(\log V)$ 。因此，更新距离并插入元素这一操作的总时间复杂度为 $O(E \log V)$ 。

将初始化阶段、取出最小元素和更新距离并插入元素的时间复杂度相加，得到： $O(V) + O(V \log V) + O(E \log V)$ 。当 V 和 E 足够大时， $O(V)$ 相对于 $O(V \log V)$ 和 $O(E \log V)$ 可以忽略不计。因此，基于堆优化的 Dijkstra 算法的总时间复杂度为 $O((V + E) \log V)$ 。

4.3.2 单源最短路径算法——Bellman-Ford算法

算法简介

刚刚我们已经掌握了 Dijkstra 算法用于求解非负权图的最短路径问题。但当图中出现负权边时，Dijkstra 算法便不再适用。此时，**Bellman-Ford算法**脱颖而出，它不仅能够处理带有负权边的图，还能检测图中是否存在负权回路。接下来，我们将深入学习这一重要算法。

Bellman-Ford算法（贝尔曼-福特算法）基于松弛操作（relaxation operation），通过对图中所有边进行多次迭代更新，逐步逼近从源点到各个顶点的最短路径。其核心思想是：每一次迭代都尝试通过其他顶点来更新到目标顶点的距离，如果找到更短的路径，就更新该顶点的距离值。经过 $V - 1$ 次迭代后（ V 为图中顶点的数量），若没有再发生距离更新，则说明已经找到了从源点到所有可达顶点的最短路径；若在 $V - 1$ 次迭代后仍能更新距离，那就意味着图中存在负权回路。

算法详解

1. 初始化

给定一个有向图 $G=(V, E)$ ，其中 V 是顶点集合， E 是边集合，指定源点为 s 。创建一个数组 $dist$ ，用于存储源点到每个顶点的最短距离估计值，初始时，将源点到自身的距离设为0，即 $dist[s] = 0$ ，将其他所有顶点到源点的距离设为无穷大。

对于图中的每条边 (u, v, w) （表示从顶点 u 到顶点 v ，边权为 w ），不进行任何操作，等待后续迭代更新。

2. 迭代过程

进行 $V - 1$ 次迭代，这里的 V 是图中顶点的总数。每次迭代都遍历图中的每一条边 (u, v, w) 。

对于每一条边，尝试进行松弛操作。具体来说，如果 $dist[u] + w < dist[v]$ （即通过顶点 u 到达顶点 v 的距离比当前记录的 $dist[v]$ 更短），则更新 $dist[v] = dist[u] + w$ 。这一步的意义在于，当发现通过其他顶点中转到 v 的路径更短时，就用这个更短的路径距离来更新 v 的距离估计值。

3. 检查负权回路

在完成 $V - 1$ 次迭代后，再进行一次边的遍历。

若在这次遍历中，仍然存在某条边 (u, v, w) 满足 $dist[u] + w < dist[v]$ ，则说明图中存在负权回路。因为如果不存在负权回路，经过 $V - 1$ 次迭代后，所有顶点的最短距离应该已经确定，不会再出现更短的路径。如果检测到负权回路，就说明从源点到某些顶点的最短路径是不存在的（因为可以沿着负权回路无限减小路径长度）。

在掌握了Bellman-Ford 算法的基本原理和操作步骤后，我们通过一个经典竞赛题目来加深对算法的理解与应用。



拓展知识

Johnson最短路

Johnson 算法的核心思想是通过重新赋权的方式，将一个可能包含负权边的图转换为一个不包含负权边的图，然后对每个顶点使用 Dijkstra 算法来求解最短路径。具体来说，它引入了一个虚拟源点，使用 Bellman - Ford 算法从该虚拟源点出发计算到其他所有顶点的最短路径，利用这些最短路径来对原图的边进行重新赋权，使得新图中所有边的权值都变为非负，这样就可以使用高效的 Dijkstra 算法来计算。

例题分析

【题目】

给定一个 n 个点的有向图，请求出图中是否存在从顶点 1 出发能到达的负环。
(负环的定义是：一条边权之和为负数的回路。)

【输入格式】

本题单测试点有多组测试数据。输入的第一行是一个整数 T ，表示测试数据的组数。对于每组数据的格式如下：第一行有两个整数，分别表示图的点数 n 和接下来给出边信息的条数 m 。接下来 m 行，每行三个整数 u, v, w 。若 $w \geq 0$ ，则表示存在一条从 u 至 v 边权为 w 的边，还存在一条从 v 至 u 边权为 w 的边。若 $w < 0$ ，则只表示存在一条从 u 至 v 边权为 w 的边。

【输出格式】

对于每组数据，输出一行一个字符串，若所求负环存在，则输出 YES，否则输出 NO。

【分步解析】

1. 定义边的结构体：定义了一个名为 Edge 的结构体，用来表示图中的边。该结构体包含三个成员变量：

from: 表示边的起始顶点。

to: 表示边的终止顶点。

weight: 表示边的权值。

结构体的构造函数 Edge(int f, int t, int w) 用于初始化这三个成员变量。

```
// 定义边的结构体
struct Edge {
    int from;
    int to;
    int weight;
    Edge(int f, int t, int w) : from(f), to(t), weight(w) {}
};
```

图4.3.5 定义边的结构体

2. 初始化距离数组

dist 是一个大小为 $n + 1$ 的向量，用于存储从源点到各个顶点的最短距离估计值。初始时，将所有顶点的距离估计值设为 INT_MAX（表示无穷大）。因为源点是顶点 1，所以将 dist[1] 设为 0。代码如图4.3.6所示。

```
vector<int> dist(n + 1, INT_MAX);
dist[1] = 0; // 源点为顶点1
```

图4.3.6 初始化距离数组

3. 进行 $n - 1$ 次迭代松弛

外层循环 `for (int i = 0; i < n - 1; ++i)` 控制迭代次数，一共进行 $n - 1$ 次迭代，这里的 n 是图中顶点的数量。在一个具有 n 个顶点的图中，任意两个顶点之间的最短路径最多包含 $n - 1$ 条边，所以进行 $n - 1$ 次迭代就可以找到最短路径。

内层循环 `for (const Edge& e : edges)` 遍历图中的每一条边。

条件判断 `if (dist[e.from] != INT_MAX && dist[e.from] + e.weight < dist[e.to])` 的作用是：首先检查源点到边的起始顶点的距离是否已经确定（不为无穷大），然后比较通过当前边到达终止顶点的距离是否比之前记录的距离更短。如果满足条件，则更新 `dist[e.to]` 的值。

代码如图4.3.7所示：

```

// 进行n - 1次迭代松弛
for (int i = 0; i < n - 1; ++i) {
    for (const Edge& e : edges) {
        if (dist[e.from] != INT_MAX && dist[e.from] + e.weight < dist[e.to]) {
            dist[e.to] = dist[e.from] + e.weight;
        }
    }
}

```

图4.3.7 进行 $n - 1$ 次迭代松弛

4. 检查负环

在完成 $n - 1$ 次迭代后，再进行一次边的遍历。

如果存在某条边 e ，使得 $dist[e.from] + e.weight < dist[e.to]$ ，这就意味着在经过 $n - 1$ 次迭代后，仍然可以找到一条更短的路径，说明图中存在负环，函数返回 `true`。

如果遍历完所有边都没有发现这样的情况，说明图中不存在负环，函数返回 `false`。代码如图4.3.8所示。

```

// 检查负环
for (const Edge& e : edges) {
    if (dist[e.from] != INT_MAX && dist[e.from] + e.weight < dist[e.to]) {
        return true; // 存在负环
    }
}

return false; // 不存在负环

```

图4.3.8 检查负环

5. 主函数处理多组测试数据

首先读取测试数据的组数 T 。

对于每组测试数据：读取顶点数 n 和边数 m 。读取每条边的信息，将其存储在 $edges$ 向量中。如果边的权值 w 大于等于 0，则添加一条反向的边，以处理双向边的情况。

调用 Bellman-Ford 函数检测图中是否存在负环，并根据返回结果输出 YES 或 NO。

代码如图4.3.9所示：

```

int main() {
    int T;
    cin >> T; // 测试数据组数

    while (T--) {
        int n, m;
        cin >> n >> m; // 顶点数n和边数m

        vector<Edge> edges;
        for (int i = 0; i < m; ++i) {
            int u, v, w;
            cin >> u >> v >> w;
            edges.push_back(Edge(u, v, w));
            if (w >= 0) {
                edges.push_back(Edge(v, u, w)); // 添加双向边
            }
        }

        if (bellmanFord(n, edges)) {
            cout << "YES" << endl;
        } else {
            cout << "NO" << endl;
        }
    }

    return 0;
}

```

图4.3.9 主函数处理多组测试数据



思考辨析

为什么Dijkstra在负权图上不可用呢？

Dijkstra 算法基于贪心思想，在处理负权图时会出现问题，主要原因有以下两点：

1、无法保证找到全局最优解

①Dijkstra 算法在运行过程中，一旦一个顶点被标记为已访问，就认为找到了从源点到该顶点的最短路径，不会再对其进行更新。

②但在负权图中，后续可能会通过负权边到达已访问的顶点，从而得到更短的路径。如果不重新考虑这些已访问的顶点，就无法找到全局最优解。

2、可能陷入无限循环

①当图中存在负权回路（即回路中边的权值之和为负数）时，Dijkstra 算法可能会陷入无限循环。

②因为负权回路会使总权值不断减小，算法可能会不断尝试沿着负权回路更新路径，导致无法终止。

4.3.3 全局最短路径——Floyd-Warshall 算法

算法简介

在图论的众多算法中，最短路径问题一直是核心问题之一。此前我们学习了 Dijkstra 算法用于求解单源最短路径问题（从一个特定源点到其他所有顶点的最短路径），以及 Bellman - Ford 算法用于处理包含负权边的单源最短路径问题和检测负权回路。然而，在某些实际场景中，我们可能需要求解图中任意两个顶点之间的最短路径，这就是全局最短路径问题。Floyd - Warshall 算法便是专门用于解决此类问题的经典算法。

Floyd - Warshall 算法基于动态规划的思想。其核心原理是通过不断引入中间顶点，尝试更新任意两个顶点之间的最短路径。具体来说，对于图中的任意两个顶点 i 和 j ，我们会依次考虑是否可以通过某个中间顶点 k 来缩短它们之间的距离。如果通过顶点 k 中转，即从 i 到 k 再到 j 的距离比当前记录的从 i 到 j 的距离更短，那么就更新这个最短距离。

算法详解

1. 初始化

给定一个有 n 个顶点的图 $G=(V, E)$ ，其中 V 是顶点集合， E 是边集合。创建一个 $n * n$ 的二维数组 $dist$ ，用于存储任意两个顶点之间的最短距离。

对于数组 $dist$ ，如果顶点 i 到顶点 j 有直接相连的边，且边的权值为 w ，则 $dist[i][j] = w$ ；如果 $i = j$ ，则 $dist[i][j] = 0$ ；如果顶点 i 到顶点 j 没有直接相连的边，则 $dist[i][j] = \infty$ 。

2. 迭代更新

进行 n 次迭代，每次迭代选择一个中间顶点 k (k 从 1 到 n)。

对于每一对顶点 i 和 j (i 从 1 到 n , j 从 1 到 n)，检查是否可以通过顶点 k 来缩短从 i 到 j 的距离。如果 $dist[i][k] + dist[k][j] < dist[i][j]$ ，则更新 $dist[i][j] = dist[i][k] + dist[k][j]$ 。

3. 结果判断

经过 n 次迭代后，数组 dist 中存储的就是任意两个顶点之间的最短距离。

同时，还可以通过检查 $dist[i][i]$ 是否为负数来判断图中是否存在负权回路。如果存在某个 i 使得 $dist[i][i] < 0$ ，则说明图中存在负权回路。

例题分析

【题目描述】

给出一张由 n 个点 m 条边组成的无向图，求出所有点对 (i,j) 之间的最短路径。

【输入格式】

第一行为两个整数 n,m，分别代表点的个数和边的条数。接下来 m 行，每行三个整数 u,v,w，代表 u,v 之间存在一条边权为 w 的边。

【输出格式】

输出 n 行每行 n 个整数。第 i 行的第 j 个整数代表从 i 到 j 的最短路径。

【分步解析】

1. 初始化矩阵距离

创建一个 $n \times n$ 的二维向量 dist 作为距离矩阵，用于存储任意两点之间的最短路径长度。初始时，将所有元素设为 INF（这里使用 INT_MAX / 2 避免后续相加时溢出），表示两点之间初始距离为无穷大，即不可达。

对于矩阵的对角线元素 $dist[i][i]$ ，将其设为 0，因为一个点到自身的距离显然为 0。代码如图4.3.10所示。

```
vector<vector<int>> dist(n, vector<int>(n, INF));
for (int i = 0; i < n; ++i) {
    dist[i][i] = 0;
}
```

4.3.10 初始化矩阵距离

2. 读取边的信息并更新距离矩阵

读取 m 条边的信息，每条边包含起点 u、终点 v 和边权 w。由于输入的点编号通常从 1 开始，而代码中数组索引从 0 开始，所以将 u 和 v 减 1。

对于无向图，边是双向的，所以同时更新 $\text{dist}[u][v]$ 和 $\text{dist}[v][u]$ 的值，取当前值和新边权 w 中的较小值。代码如图4.3.11所示。

```
for (int i = 0; i < m; ++i) {
    int u, v, w;
    cin >> u >> v >> w;
    u--;
    v--;
    dist[u][v] = min(dist[u][v], w);
    dist[v][u] = min(dist[v][u], w);
}
```

4.3.11 读取边的信息并更新距离矩阵

3. 核心的 Floyd - Warshall 算法部分

最外层循环的变量 k 表示中间节点，它尝试所有可能的中间节点。内层的两层循环 i 和 j 遍历所有的点对。

对于每一个点对 (i, j) ，检查是否可以通过中间节点 k 使得从 i 到 j 的路径更短。如果 $\text{dist}[i][k]$ 和 $\text{dist}[k][j]$ 都不是无穷大（即 i 到 k 和 k 到 j 都可达），并且 $\text{dist}[i][k] + \text{dist}[k][j]$ 小于当前记录的 $\text{dist}[i][j]$ ，则更新 $\text{dist}[i][j]$ 的值为 $\text{dist}[i][k] + \text{dist}[k][j]$ 。代码如图4.3.12所示。

```
for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] +
                dist[k][j] < dist[i][j]) {
                dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
}
```

4.3.12 核心算法部分

4. 输出结果

遍历距离矩阵 dist ，如果 $\text{dist}[i][j]$ 仍然是 INF ，说明从 i 到 j 不可达，输出 INF ；否则输出 $\text{dist}[i][j]$ 的值。代码如图4.3.13所示。

```

for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        if (dist[i][j] == INF) {
            cout << "INF ";
        } else {
            cout << dist[i][j] << " ";
        }
    }
    cout << endl;
}

```

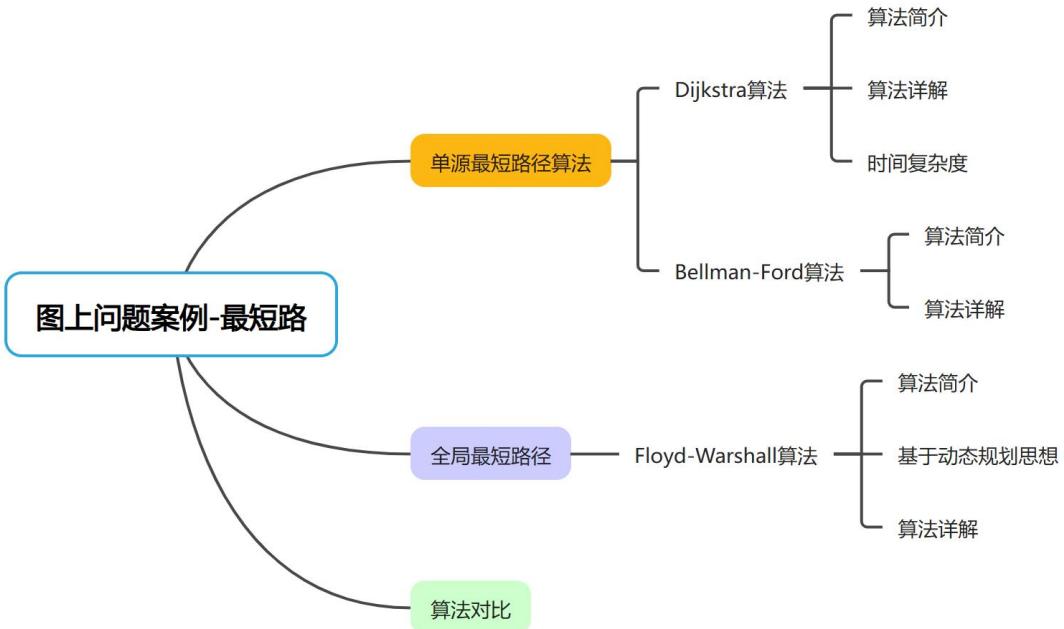
4.3.13 输出结果



知识链接：算法对比

我们已经学习了基于堆优化的 Dijkstra 算法、Bellman - Ford 算法和 Floyd - Warshall 算法。这三种算法各有特点，在不同的场景下有着不同的表现。下面我们将详细对比它们的时间复杂度和适用场景。

算法	时间复杂度	使用场景
基于堆优化的 Dijkstra 算法	$O((V + E)\log V)$	非负权图的单源最短路径问题，稀疏图
Bellman - Ford 算法	$O(VE)$	包含负权边的单源最短路径问题，检测负权回路
Floyd - Warshall 算法	$O(V^3)$	全局最短路径问题，顶点数较少的图


 练习提升

- 某外汇交易平台提供货币兑换汇率表。例如，1美元兑换0.9欧元，1欧元兑换110日元，1日元兑换0.01美元。设计一个算法，判断是否存在通过循环兑换获利的可能（即负权环）。

【输入】

货币种类数 n 和兑换关系数 m 。接下来 m 行，每行为货币A与货币B汇率，表示1单位A可兑换的B的数量。

【输出】

“存在套利机会”或“无套利机会”。

- 在社交网络中，若两个人的最短好友链长度 ≤ 6 ，则称他们满足“六度空间”理论。给定好友关系图（无向无权图），计算所有点对中最短路径的最大值，验证是否 ≤ 6 。

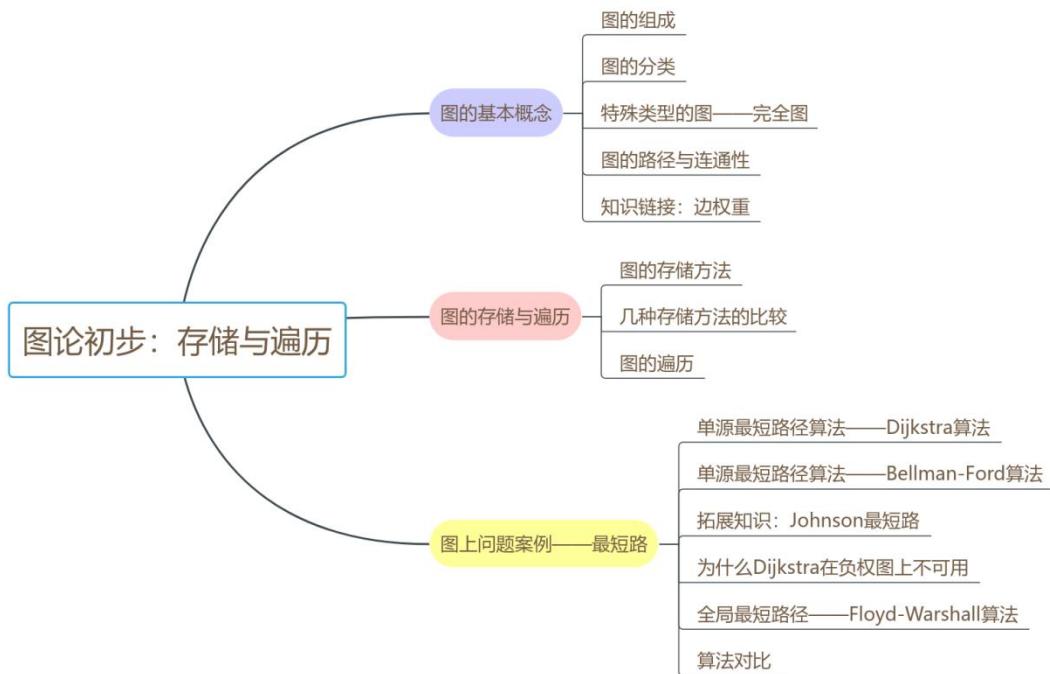
【输入】

用户数 n 和好友关系数 m 。接下来 m 行，每行为 uv ，表示用户 u 和 v 是好友。

【输出】

最长的最短路径长度。若 > 6 ，输出“不符合六度空间”，否则输出“符合”。

1. 下图展示了本章的核心概念与关键能力，请同学们对照图中的内容进行总结。



2. 根据自己的掌握情况填写下表。

学习内容	掌握程度
图的基本概念与术语	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
图的存储结构	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
图的遍历算法	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
最短路径算法	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
负权环检测	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
完全图与稀疏图的特点及适用场景	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解
图的动态操作	<input type="checkbox"/> 不了解 <input type="checkbox"/> 了解 <input type="checkbox"/> 理解

小结

在本章的学习过程中，我们首先深入探索了图的基本概念。从顶点、边、权值这些构成图的核心元素出发，我们认识到，顶点就像是现实世界中的一个个实体，边则代表着实体之间的关系，而权值能够量化这些关系的某些属性。通过对无向图和有向图等不同类型图的特点分析，我们明白了图结构在描述不同关系时的灵活性和多样性，这是构建图论知识体系的重要基石。

随后，我们聚焦于图的存储方式，深入学习了邻接矩阵和邻接表这两种重要方法。邻接矩阵以二维数组的形式，直观地展现了顶点之间的连接关系，在判断两点是否相连时效率极高，但是在处理稀疏图时，会造成大量的空间浪费；邻接表则通过链表或动态数组的方式，将与每个顶点相连的边存储起来，这种方式在存储稀疏图时优势明显，能够有效节省空间。我们明白了，根据实际问题的特点和数据规模，合理选择存储结构，是提升程序运行效率的关键。

在图的遍历环节，广度优先搜索（BFS）和深度优先搜索（DFS）为我们提供了两种截然不同却又同样高效的探索方式。BFS 按照距离起始顶点由近到远的顺序访问节点，常用于寻找最短路径等问题；而 DFS 则沿着一条路径不断深入，直到无法继续才回溯，这种方式在寻找连通分量等问题上表现出色。掌握它们的逻辑与实现方法，我们就拥有了在图结构中自由穿梭、获取数据的有效途径。

最后，我们通过最短路问题这个经典案例，将前面所学的理论知识应用于实践。迪杰斯特拉（Dijkstra）算法和弗洛伊德（Floyd）算法的学习与实践，让我们深刻体会到了图论算法的精妙之处。Dijkstra 算法适用于边权非负的图，通过不断选择距离源点最近的未确定顶点，逐步扩展最短路径；Floyd 算法则基于动态规划思想，能够一次性求出图中任意两点之间的最短路径。这些算法不仅帮助我们解决了具体的问题，更让我们学会了如何将理论转化为实际的代码实现。

图论知识不仅是 CSP-J 竞赛的重要考点，更是解决复杂实际问题的有力武器。它教会我们用计算机的思维，将现实世界中错综复杂的关系抽象成能够处理的数据结构，通过算法优化找到最佳解决方案。通过这一章的学习，希望同学们不仅熟练掌握图论的基本概念和算法，更要注重培养抽象建模和逻辑思维能力。

后记

本册教材《初中生C++竞赛宝典：数据结构实战（J组专用）》由华中师范大学信息学竞赛教研组联合多省市一线竞赛教练共同编写，旨在为初中生CSP-J竞赛选手提供系统化、实战化的数据结构学习指南。教材内容严格遵循中国计算机学会（CCF）竞赛大纲要求，结合十余年竞赛辅导经验，充分吸纳往届优秀选手的实战反馈，力求将抽象的理论知识与经典赛题深度融合，助力学生夯实基础、提升算法思维。

本教材的编写凝聚了教育工作者、信息学竞赛专家、专业程序员及资深教材编辑的集体智慧。编写团队立足初中生认知特点，以“分层递进、案例驱动”为核心理念，通过模块化知识讲解、真题剖析与拓展训练，构建了符合J组竞赛需求的学习体系。参与本册教材编写的主要成员包括：洪嘉璐（统筹）、张书颇（数据结构模块设计）、田小琴（代码实现与调试）、廖昌升（习题解析与优化），审校团队由刘志轩、吴昊、周敏等资深竞赛评委组成。教材版式设计与插图绘制由陈兴佳、李舒红完成，部分案例灵感来源于历年CSP-J/S真题及国际信息学竞赛（IOI）优秀选手的解题思路。

在此，我们衷心感谢为教材编写提供支持的学校教研组、参与试用的师生及技术社区开发者。特别感谢往届选手分享的实战经验，以及开源社区为算法可视化工具提供的资源支持。教材中部分代码框架参考了CCF官方题解思路，在此一并致谢。

教材中涉及的例题、习题及配图，我们已尽力标注来源或征得原作者同意。若存在疏漏或未署名的内容，请相关权利人及时联系编写组（邮箱：jzu_textbook@xx.edu.cn），我们将妥善处理并致谢。

信息学竞赛领域发展迅速，算法与数据结构的教学方法亦需与时俱进。我们诚挚欢迎广大师生、竞赛教练及读者在使用过程中提出宝贵建议。编写组将定期收集反馈，持续优化内容，为初中生信息学竞赛教育提供更优质的资源。

联系方式

电 话：13646677566

电子邮箱：1469315288@qq.com

中国地图出版社教材出版分社

人民教育出版社课程教材研究所信息技术课程教材研究开发中心

2025年5月

J组专用辅导书

初中生C++竞赛宝典

数据结构实战

J组专用辅导书
J ZU ZHUANYONG FUDAOOSHU

初中生C++竞赛宝典：数据结构实战

CHUZHONGSHENG C++ JINGSAIBAODIAN SHUJUJIEGOUISHIJIAN

洪嘉璐 主编

责任编辑 张书颇 美术编辑 陈兴佳

责任校对 洪嘉璐 封面设计 陈兴佳

责任印务 蒋玲 插画 李舒红

出版发行 第九教育出版社
(武汉市雄楚大道152号 邮编: 430079)

图文制作 信息技术课程教学设计第九组

印 制 华中师范大学
开 本 890mm×1240mm 1/16

印 张 10.5
字 数 242500
版 次 2025年5月第1版
印 次 2025年5月第1次印刷
标准书号 ISBN 978-7-5536-0000-0
定 价 50.00元